NATURAL

Natural

Frame Gallery

Version 5.1.1 for Windows

software AG

This document applies to Natural Version 5.1.1 for Windows and to all subsequent releases. Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Table of Contents

Frame Gallery - Overview

Frame Gallery - Overview

The Natural Frame Gallery documentation describes features of Frame Gallery, the Natural development environment which you will use on a daily basis to create and maintain applications. First, you are introduced to the landscape of the graphical user interface so that you always find what you need exactly when you need it. Then you are provided with a task-oriented description of each of the major editors used to create applications: the program editor, data area editor, map editor, DDM editor, dialog editor, and report writer.

Using the frame gallery, you can select one of a range of standard dialog types (or frames) and generate a simple dialog and associated data access modules. You can then use the dialog editor to customize the generated dialog to include application-specific validation and other processing needed for a fully functional application. Suggested code in the generated dialog assists you with the customization process.

You can use frame gallery to generate prototype dialogs and functions during the product design phase to see what an application will eventually look like. Frame gallery is primarily used, however, to generate dialogs and functions during the implementation phase of a software project.

- Frame Gallery - General Information
- Designing the User Interface
- Frame Gallery Naming Conventions
- Designing the Application Structure
- Selecting Frames
- Using Tables in Frame Gallery
- Generating Functions in Frame Gallery
- Customizing a Generated Application
- Communication Between Dialogs
- Application Frames
- Standard Commands
- Customizable Components
- Reusable Components
- Background Processes
- The Command System
- List Box Handling
- Creating Object Views
- Data Storage and Data Access
- Transaction Logic
- Data Transfer
- Locking Logic
- Creating an SQL Access Layer
- User Exits
- Business Specification Descriptions

The authors of the Frame Gallery documentation assume that you have a working knowledge of Microsoft Windows and the terminology used to describe it. If not, consult the Windows documentation for a description of basic Windows elements, usage and terminology.

Platform-Specific Information

Wherever necessary, platform-specific information in the present documentation is identified by the following terms:

Mainframe Refers to the operating systems OS/390, VSE/ESA, VM/CMS and BS2000/OSD, as well as all TP monitors supported by Natural under these operating systems.

OpenVMS Refers to the OpenVMS operating system.

UNIX Refers to all UNIX systems supported by Natural.

Windows Refers to the following operating systems:

In a Natural development environment:

- Microsoft Windows NT
- Microsoft Windows 2000

In a Natural run-time environment:

- Microsoft Windows 98
- Microsoft Windows NT
- Microsoft Windows 2000

OS/400 Refers to the OS/400 operating system running on AS/400 and iSeries 400 machines. See the documentation provided on the Natural for OS/400 product CD-ROM.

Frame Gallery - General Information

The following topics are covered below:

- What is the Frame Gallery?
- Benefits Provided by Frame Gallery
- Application Development Procedure

What is the Frame Gallery?

The frame gallery and the application shell are development tools for dialog applications. The main components of the frame gallery and the application shell are:

- **Frame gallery**
  Using the frame gallery, you can select one of a range of standard dialog types (or frames) and generate a simple dialog and associated data access modules. You can then use the dialog editor to customize the generated dialog to include application-specific validation and other processing needed for a fully functional application. Suggested code in the generated dialog assists you with the customization process. You can use frame gallery to generate prototype dialogs and functions during the product design phase to see what an application will eventually look like. Frame gallery is primarily used, however, to generate dialogs and functions during the implementation phase.
- **Application frames**
  The frames used in the frame gallery include large amounts of standard internal logic to handle browsing, command processing, navigation between dialogs and database access. In addition to these frames, further frame code is available which you can include in your application to handle various special requirements.
- **Application shell**
  The application shell provides an application framework which you can use to run dialogs and applications developed using the frame gallery. It also provides a range of administrative functions used in application development (for example, application definition) as well as in maintenance (for example, user maintenance).

Benefits Provided by Frame Gallery

The frame gallery contains all components of a dialog system that are not application-specific. Into this generic system you add your application-specific functionality.

Using frame gallery provides the following advantages over conventional application development:

- Reduced implementation requirements.
  Standard functions need not be individually coded for each new application. The developer can concentrate on the often very complex application-specific requirements.
- Reduced testing requirements.
  The frames provided have already been tested and are error free.
- Easy customization to meet application-specific requirements.
  In addition to basic functionality, most frames also contain suggested code which can be used to customize dialog functions to meet application-specific requirements.
- Protection of investment in application-specific code.
  Standardized logic and application-specific code are carefully separated in dialogs through the extensive use of copy code for the standardized frame logic. This means that upgrades to frame logic in future versions of the Frame Gallery can be easily incorporated by simply restoring existing dialogs.
- Easy system orientation and maintenance for developers. Because a similar structure is used for all frames, it is easier for all developers to become acquainted with any given functional aspect of the application. This

reduces orientation and maintenance requirements significantly.

- Standard application navigation for end users. End users are always provided with the same applications structure, which increases system acceptance by the user while at the same time reduces user orientation and training requirements.

Application Development Procedure

The purpose of this section is to describe how an application could be created using application shell and frame gallery. The information provided is meant to be a guideline and not a complete recipe for application development.

The procedure below is a general recommendation for creating applications.

1. Write application specification and analyze requirements.
2. Define user interface standards and naming conventions.
3. Define and/or generate database schema and database definition. Decide which entities are to be implemented as tables.
4. Familiarize yourself with application shell and frame gallery.
5. Structure the application by determining what types of dialogs and functions are necessary.
6. Assign a frame to each function.
7. Define a start application in the application shell and specify a Natural library for the application.
8. Generate prototype dialogs and functions.
9. If not already performed in step 3, define and/or generate database schema and database definition. Decide which entities are to be implemented as tables.
10. Define tables.
11. Generate production dialogs and functions.
12. Customize the application.
13. Set up icon-based navigation.

Designing the User Interface

This section contains recommendations for ergonomic user interface design.

- Standard Layout Settings
- Push Button Spacing
- List Boxes
- Selection Boxes and Combo Boxes
- Menu Bars

For more specific information, please refer to the *SOFTWARE AG GUI Style Guide* or other published graphical user interface style guides.

To ensure that a your application conforms to guidelines of ergonomic design, you should define user interface standards at the earliest possible phase of the project, prior to application prototyping.

It is best to apply the same standards to all applications so that they have a consistent appearance.

You should design the end-user interface in co-operation with those individuals who will eventually use the application. The ultimate acceptance of an application is directly related to the degree of co-operation with end users during the inception as well as during subsequent stages.

Standard Layout Settings

Frames are provided with standard layout settings. For example:

- Dialog size for application windows (MDI Frame) 640*480 pixel
- Dialog size for maintenance functions (MDI Child) 630*320 pixel
- The grid for the Natural editor is set to 5*5 pixels and the attribute Snap into Grid is marked.

Note:
Each dialog element is accessible via an access key ID. This access key ID must be unique within a dialog. Elements which are used frequently should be assigned the same access key ID for all dialogs.

Push Button Spacing

The following spacing is used between push button areas:

- For push buttons positioned along the right margin:
  - between push buttons of the same group 5 pixel
  - between groups 15 pixel
  - between push buttons and dialog border 10 pixel (minimum)
- For push buttons positioned along the bottom margin:
  - between push buttons of the same group 10 pixel
  - between groups 20 pixel
  - between push buttons and dialog border 10 pixel (minimum)

List Boxes

For ergonomic reasons, the number of visible entries in a list box is limited to 8.

Selection Boxes and Combo Boxes

The number of visible entries in the list portion of such boxes is limited to 6.

Menu Bars

For dialogs produced using the frame gallery, all menu entries required for a normal dialog are predefined. If these are not sufficient, the existing menus can be modified.

The following table contains the standard frame gallery menus together with the frame gallery internal variable names and short description.

Object (Z_OBJECT)	All common menu entries required for graphical user interface applications, plus all actions which result in database access. If the space available is not sufficient your special requirements, an additional menu can be created. The name of the additional menu could be COMMAND.
Edit (Z_EDIT)	Menu for implementing all actions related to editing (processing) of the object which has just been read. For example, copy or paste functions.
View (Z_VIEW)	Menu for all zoom functions. Subdialogs and modal windows for a main dialog can be assigned here.
Selection (Z_SELECT)	Menu used for navigation within an application.
Options (Z_OPTIONS)	Menu for maintaining profiles. (To be available with the next release.)
Window (Z_WINDOW)	Standard menu for graphical user interface applications which contains entries with which the current open dialog can be arranged on the screen, and also to indicate which dialog must have the focus.
Help (Z_HELP)	Standard menu for graphical user interface applications which provides various types of help information.

Frame Gallery Naming Conventions

This section provides information on naming conventions within frame gallery.

- Reserved Identifiers for Key Values
- Conventions for Message Text
- Natural Object Names
- Frame Gallery Object Names

Reserved Identifiers for Key Values

To distinguish frame gallery objects from those which are created during the development of an application, certain prefixes and value ranges are reserved for frame gallery objects.

The identifiers listed in the table below must not be used for any of the following:

- Names of objects of the documentation tool being used;
- Names of all Natural modules;
- Designation of variables and database fields;
- Key IDs for:
  - applications;
  - commands;
  - object types;
  - functions;
  - function groups;
  - background procedures;
  - dialog types;
  - symbol bars.

The reserved identifiers are:

Identifier	Object
Z*	Natural module names
GZ_*	Global data contained in global data areas
LZ_*	Local data contained in local data areas
PZ_*	Parameter data in parameter data areas
#Z	Local data
Z_*	Applications, commands, object types, function groups, background procedures, dialog types, tool bars
Z*	Functions

Conventions for Message Text

The application shell requires message texts. These texts are partly referenced from the frames. The following numbering scheme is used:

- 0001-0999: system messages
- 1000-9999: application messages

The application shell messages are in the file msg1.dat (English) and msg2.dat (German) respectively. These files serve as the basis for the generation of a Natural message file.

System Messages

This message area contains

- General error messages such as "data record not found";
- Error messages for the application shell.

You should not make additional entries in this message area.

Application Messages

This message area is reserved for messages used by applications implemented using the application shell.

It is recommended to classify messages for various applications by assigning specific ranges to each application. This avoids possible conflicts in message number usage among applications.

Natural Object Names

This section describes the naming conventions for Natural objects as well as the designation of program variables and database fields. These naming conventions are general recommendations for the designation of individual objects.

Structure

Natural objects which do not contain any language-dependent components must be named according to the following structure:

BOOXXXXN

where *B* is the business area abbreviation (for example, P for Purchasing), *OO* is the object type abbreviation, *XXXX* is the function identifier and *N* is the Natural object type.

Natural objects which are language-dependent must be named according to the following structure:

BOOXXXLN

where *B* is the business area abbreviation, *OO* is the object type abbreviation, *XXX* is the function identifier, *L* is the language code and *N* is the Natural object type.

Business Area Abbreviation

The abbreviation of the business area to which the function belongs. Such abbreviations must be set organization-wide to ensure uniqueness and consistency.

Examples:

P	Purchasing
S	Sales
I	Inventory

Object Type Abbreviation

The abbreviation of the object type to be processed.

If the Natural module is used together with a specific information object, the abbreviation of the information object must be as descriptive as possible.

Example:

CU	for Customer

If the Natural module is used for a specific central task, the first position of the object type abbreviation must contain the character (X) and the second position must contain an abbreviation of the processing task.

Example:

XD	for Dating

If the Natural module is of primary importance to the entire application and can be used throughout the application, the object type abbreviation must be set to XX.

Function Identification

An abbreviation which identifies the function uniquely. If the module is a central, frequently reused module, it is important to use an abbreviation which is as descriptive as possible.

If the module is used only for a specific information object or processing task, the following guidelines can be used:

Positions 1 and 2	Type of Processing
AM	Multiple record access module
AS	Single record access module
BR	Listing of objects for an object type (browse dialog)
DD	Delete (deletion subprogram)
KD	Key ID for dialog (Key dialog)
MD	Maintain processing function (maintain dialog)
MR	Multiple records (mass processing dialog)
MW	Modal window processing (modal window)
SU	Lower level window (subdialog)
NS	Individual dialog (nonstandard dialog)

Positions 3 (and possibly 4) are used as additional numeric qualifiers (01-99) in the event that there is more than one module per type of processing. Otherwise, these positions may be set to zero.

Natural Language Code

The code of the language belonging to the interface of the Natural module.

If the system is multilingual, you must specify the special character which is to replace the current language code at runtime.

Natural Object Type

The Natural object type. Examples are shown below.

- A = parameter
- C = copycode
- D = dialog
- G = global data area
- L = local data area
- N = subprogram
- P = program
- S = external subroutine
- T = text

Please note the following difference in dialog designations between Natural and frame gallery:

- In Natural, a dialog is designated by Natural object type ’3’.
- In frame gallery, a dialog is designated by ’D’.

Examples:

TXXINF1D

T	Business area: T for Travel Cost Reimbursement
XX	Internal, reusable module
INF	Information display
1	Language code: 1 for English
D	Natural object type: D for dialog

PCUMD02D

P	Business area: P for Purchasing
CU	Object type abbreviation: CU for Customer
MD	Function abbreviation: MD for main dialog
0	Indicates that this is the only such module for this area
2	Language code: 2 for German
D	Natural object type: D for dialog

SORAS00N

P	Business area: S for Sales
OR	Object type abbreviation: OR for Order
AS	Function abbreviation: AS for access single record
0	Indicates that this is the only such module for this area
0	Language code: 0 indicates no language dependency
N	Natural object type: N for subprogram

Field Names

Variable Names

The general structure used for variable names is as follows:

prefix_ BOO _name _suffix

prefix	G_ global variable
	L_ local variable defined in an local data area
	P_ parameter data defined in a parameter data area
	# user-defined local variables
B	Business area in which variable is used. May be omitted if the variable is not to be associated with a specific business area.
OO	Object type for which variable is used. May be omitted if the variable is not to be associated with a specific information object.
name	The name of the variable. The name must be as descriptive as possible.
suffix	Further classification of the variable. **Examples:** _CV Control variable _FROM Variable used for starting value _THRU Variable used for ending value _A20 Variable used for field format

Handle Names

In event-driven applications, handle names are used to describe dialog elements. In order to be able to provide a clearer description of a business element used within a dialog, the following naming structure is recommended:

prefix _ control-abbreviation _BOO _ name

prefix	Variable type. In that handles are defined locally in a dialog, the character # is used here. Frame gallery-specific handles are identified with the characters #Z.
control-abbreviation	Natural-defined abbreviation for a control variable. **Examples:** **BM** - bitmap **PB** - push button **IF** - input field
B	Business area in which variable is used.
OO	Object type for which variable is used.
name	Descriptive name of the handle.

Frame Gallery Object Names

These naming conventions are general recommendations for the designation of individual objects.

General

The key ID for the application shell administration system is generally preceded with the default abbreviation Z. Therefore, user data records must not begin with Z. For further information, see Reserved Identifiers for Key Values.

Frame Gallery Function Names

In order to clearly associate a function with an application via the function key ID, the business area and object type abbreviations must precede the key ID.

Examples:

ZUS_MNT

Z	Business area abbreviation reserved for frame gallery
US	Object type abbreviation (user maintenance)
MNT	Key ID for maintenance function

SCU_DIS

S	Business area abbreviation (S for Sales)
CU	Object type abbreviation (CU for Customer)
DISP	Key ID for function which displays a data record

Designing the Application Structure

The design of dialog functions is one of the most important activities at the beginning of a design phase. It is separated into the following tasks:

- Definition of the application structure.
- Separation of the system into business functions.
- Definition of the structure of the individually callable components.
- Decision on which frames will be used for which business functions.

The following topics are covered below:

- Familiarizing Yourself with the Application Frames
- The Business Function

Familiarizing Yourself with the Application Frames

Before designing the application structure, you should make sure you are familiar with the application frames. The selected frame determines the basic functionality of a function or a processing step. Frame selection is therefore a very important step in the transition between the functional design phase and implementation.

In order to select frames, you need:

- an overall understanding of which frames are available;
- an understanding of the possible combinations of program frames which can be used when; creating a function.
- to identify reusable components (this reduces the effort involved in frame selection);
- to consider the specific functional requirements for each function.

The Business Function

A business function is the smallest individually callable application unit. It can be an administrative function for maintaining data, an action such as making a reservation, or an enquiry. Depending on requirements, it could consist of a single dialog or a main dialog and a number of subordinate dialogs.

Each dialog can be supported by additional functionality such as background processing or a sequence of further lower level windows (e.g. selection help).

- Identifying a Business Function
- Structuring a Function

Identifying a Business Function

A business function acts on a group of related data, subsequently referred to as an object. An object can contain fields from various physical data sources.

A business function can include several actions for an object (for example, modify or copy), or be limited to one action (for example, delete).

Structuring a Function

Functions can be broken down into individual dialogs. You select the frame for each dialog to be used depending on the functionality required.

You should consider the following guidelines when creating functions:

- Use as few dialogs as possible.
- Place logically related fields on the same window.
- Place mandatory fields on the maintain dialog whenever possible. If there is insufficient space, or if no logical structure is evident, then place the mandatory fields within a subordinate dialog which you indicate as important. For example, via a push button.
- Determine if a function contains lower level dialogs that are shareable with other functions. Define such common functionality as separate reusable dialogs.
- Make such reusable modules available to the project team as soon as possible. Use subprograms for related business functionality which is more of a business nature. Use subordinate dialogs or modal windows for related business functionality which must be invoked by the end user as a dialog.
- Check for functionality which can be optionally invoked by the end user, and which are not required for the normal completion of the function. Define such functionality as subordinate dialogs or as modal windows.
- Check the necessity of selection help for the individual fields. Selection help for key fields should always be available.
- Check if confirmation or information windows are to be used.
- Determine the necessary frame functionality which is required.

- Build the call mechanisms for lower level dialogs in a way which is logical when seen from the business perspective (example, push buttons in prioritized sequence).

Selecting Frames

The following guidelines are divided into those applicable for entry-level dialogs and those applicable for two lower level subordinate dialogs.

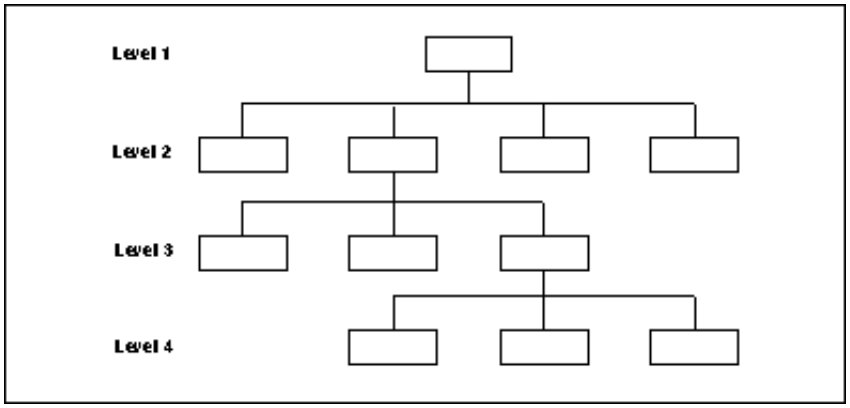
1. First select the appropriate entry-level dialog.
2. Next select the frames for the logically subordinate modules, such as subordinate dialogs, modal windows and object selection.
3. Select a frame for each window.
4. If you find alternatives, select those which appear to be the most flexible for the function.

The following topics are covered below:

- Dialog Structure
- Examples for Combining Production Frames

Dialog Structure

Each business function contains exactly one entry-level dialog from which various subordinate dialogs can be invoked. Entry-level dialogs as well as subordinate dialogs can call modal windows or key dialogs. A modal window can call additional modal windows or key dialogs. The following graphics shows the tree structure of a function with levels.



The following topics are covered below:

- Entry-Level Dialog - Level 1
- Subordinate Dialogs - Level 2
- Modal Dialogs - Level 3
- Permissible Calls

Entry-Level Dialog - Level 1

Entry level dialogs represent the first level of a function. They can be used without restriction and also stand-alone. The following frames can be used.

Display, modify, add, and operative functions

Frame	Maintain dialog
Call	Direct
Further calls to	Subordinate dialogs, key dialogs, modal windows
Special characteristics	Works with preliminary data copies

Browse, Overviews

Frame	Browse dialog
Call	Direct
Further calls to	Modal windows, mass processing dialog, key dialogs, deletion subprogram, maintain dialog
Comment:	A business function can be called by selecting an action in a browse dialog

Mass Processing

Frame	Mass processing dialog
Call	Direct
Further calls to	Modal windows, key dialogs
Comment	Recommended in conjunction with a browse dialog

Delete Confirmation

Frame:	Deletion
Call	Direct
Further calls to	None
Comment	Creates a subprogram which is activated directly from the command system

Nonstandard Dialog

Frame	Nonstandard dialog
Call	Direct
Further calls to	Modal windows, key dialogs

Subordinate Dialogs - Level 2

Key Dialog

Frame	Key dialog
Call	From maintain dialog, browse dialog
Further calls to	Modal window

Display, modify, and new functions, and other actions

Frame	Subdialog
Call	From maintain dialog
Further calls to	Key dialogs, modal windows
Special characteristics	Works with preliminary copies

Display, modify, new, operative, and browse functions without access to preliminary data

Frame	Modal window
Call	From maintain dialog, browse dialog, mass processing dialog
Further calls to	Key dialogs, modal windows
Special characteristics	Works with preliminary copies

Mass Processing

Frame	Mass processing dialog
Call	From browse dialog
Further calls to	Key dialogs, modal windows

Modal Dialogs - Level 3

Display, modify, new, and operative functions:

Frame	Modal window
Call	From mass processing dialog, key dialog
Further calls to	Key dialogs, modal windows

Key Dialog

Frame	Key dialog
Call	From subdialog, modal window, key dialog, mass processing dialog
Further calls to	Modal window
Special characteristics	Generated maintain dialogs include a call to a key dialog for input of the primary key. Key dialogs can also be used for active help and foreign-key input. In this case, you must code the call to the key dialog yourself.

Permissible Calls

The following dialogs can be called:

Called by	**Subdialog**	**Modal Window**	**Key Dialog**	**Mass Processing**
Maintain dialog	X	X	X	
Browse dialog		X	X	X
Mass processing dialog		X	X	
Nonstandard		X	X	
Deletion				
Subdialog		X	X	
Modal window		X	X	
Key dialog		X	X	

Examples for Combining Production Frames

The following topics are covered below:

- Basic Combinations
- Variations

Basic Combinations

Key input and display or modification of data in a window:

Entry Dialog	Maintain dialog
Key Selection	Key dialog

Listing of data records of an object:

Listing	Browse dialog

Mass Processing

Overview:	Browse dialog
Mass Processing	Mass processing dialog

Variations

Key input and display of several data windows:

Entry Dialog:	Maintain dialog
Key Selection:	Key dialog
Display Window	Subdialog

Key input and display of several data windows with foreign key selection:

Entry Dialog	Maintain dialog
Key Selection:	Key dialog
Display Window	Subdialog
Foreign Key Selection	Key dialog

Input of selection criteria for a list of data records, selection of a key from a list and display of multiple data windows:

Entry Dialog 1	Browse dialog

This is created from two independent functions:

- the browse dialog, and
- display functions.

The display function can be invoked using a command from the browse dialog. Selected key values are passed.

Entry Dialog	Maintain dialog
Key Selection	Key dialog
Display Window	Subdialog
Selection Window	Modal window

The selection processing can be used for the selection of data as well as to determine subsequent processing.

Input of a selection criterion to determine subsequent processing:

Entry Dialog	Nonstandard dialog
Subsequent Processing	Subdialog

Listing with optional expanded selection in a subordinate window:

Listing	Browse dialog
Selection	Modal window or subdialog

Mass Processing

Overview	Browse dialog
Mass Processing	Mass processing dialog
Zoom	Modal window

Note:
All data modifications are immediately applied to the database. No preliminary copies are created.

Batch Programs

Call from an online function:

Batch Program:	Skeleton

The batch program can be called from any module.

Alternate batch program:

Selection:	Nonstandard dialog
Batch Program:	Skeleton

This variant is recommended when, for example, several optional lists are to be created as batch programs.

Using Tables in Frame Gallery

The following topics are covered below:

- Using Tables
- Creating Help for Table Data
- Creating an Access Module for a Table
- Creating a User Exit for Single-object Processing

Using Tables

Codes are frequently used in applications to shorten input and/or to guarantee the uniqueness of the texts or names allocated to these codes. These codes are generally stored in tables. Tables can also be used, for example, to make multiple-language texts available.

These tables are rarely modified, for example:

- Status table:
  1 = in development
  2 = in test
  3 = released
- Country table:
  D = Germany
  CH = Switzerland

Even a currency table which is updated daily is considered a table in this context.

No special actions, for example, bookings or calculations, are carried out on table data. Only maintenance functions, such as "add", "modify" and "delete", are necessary to update the content.

Otherwise, the table data are used only for display, for validation or as a basis for calculations when processing other objects.

The following topics are covered below:

- Criteria for Defining a Table
- Maintenance Functions for Tables
- Access to Table Data
- Selection Help for Table Data

Criteria for Defining a Table

To define an entity as an application shell table, the following criteria must be met:

- There are at most 50 fields, including the table keys.
- There are alphanumeric and numeric key fields.
- There can be up to four keys.
- The table key can consist of up to five components.
- No numeric field is longer than 27 digits.
- No alphanumeric field is longer than 240 characters.
- The table key is not longer than 30 characters.
- Processing is limited to the maintenance functions Add, Modify, Display and Delete.

Maintenance Functions for Tables

Tables of the kind described above can be defined and maintained with the help of the application shell table administration system.

No file must be created, and the implementation of maintenance functions is also not necessary. All necessary maintenance functions are available immediately after the definition of the table in the application shell.

If special validation is necessary for the processing of the table data, the user exit of the table-administration system can be used to call specific processing for the table.

How to create tables is described in the Natural Application Shell Manual.

Access to Table Data

To access table data from your application functions, access modules are available, which are described in section Creating an Access Module for a Table.

Selection Help for Table Data

A module for carrying out selection help through table keys, is available. This module is parameter driven and is callable from any application dialog function.

Creating Help for Table Data

Input for a function must often be validated against a table. Active help in the form of a selection table ensures that the end user enters a valid value.

To link selection help for table data into a dialog, you have to code the call to the selection help and the receipt of the selected value.

For integration of a selection help for table data in an individual dialog, you must code the call of the selection help and the receipt of the selected value.

Define the following local variables:

```
1 #REF_TAB_SEL_INFO (A65)
1 REDEFINE # REF_TAB_SEL_INFO
2 #REF_TAB_CLIENT_ID    (A2)
2 #REF_TAB_ID        (A12)
2 #REF_TAB_DESC_NUM    (N1)
2 #REF_TAB_VALUE        (A30)
```

Invoke the selection help using the following suggested code:

```
MOVE LZ_KEY_TYPE_FOREIGN TO LZ_KEY_TYPE
*
MOVE client ID        TO #REF_TAB_CLIENT_ID
MOVE table ID        TO #REF_TAB_ID
MOVE descriptor number       TO #REF_TAB_DESC_NUM
MOVE field value        TO #REF_TAB_VALUE
MOVE #REF_TAB_SEL_INFO TO PZ_LOCAL.PZ_SEL_KEY
*
OPEN DIALOG 'ZCAKDFD' #DLG$WINDOW WITH PZ_LOCAL
```

Receive the selected value in customizable component Z_RECEIVE_KEY:

```
MOVE PZ_RECEIVE.PZ_SEL_KEY  TO #REF_TAB_SEL_INFO
MOVE #REF_TAB_VALUE         TO ...
```


Creating an Access Module for a Table

The interface of the general access module for table data is large and complex. It is therefore useful - in order to access table data from a business function - to create an access module especially designed for a single table, which returns the required fields, and is easy to use.

The following steps are recommended:

1. Use an existing module as basis.
2. Save it under a different name.
   Do not modify the example module.

The following naming convention is used for access modules:

xxxAS00y

where *xxx* is the object code and *y* is the Natural object type.

Example	Library	Save as
ZXFCAS0A (PDA)	SYSCOMP	xxxAS00A (as parameter data area)
ZXFCAS0N (Text)	SYSCOMP	xxxAS00N (as subprogram)
ZXFCAS0D (Dialog)	SYSCOMP	xxxAS00D (as dialog)

Parameter Data Area

Enter the fields for the table using exact formats and lengths.

Warning:
You must not modify the field P_PTS nor the lengths of the key fields.

Subprogram

Complete the place holder for parameter data area, field names, table names, client ID and language code according to the instructions in the subprogram.

Dialog

The dialog can be used to test the subprogram.

Include the parameter data area you have just created (see section Parameter Data Area) into the local data definition.

Adapt the user interface as follows:

- include an input field for each table field
- link the input field via linked variables to the corresponding variable from the parameter data area
- specify a text constant for each input field.

Warning:
Do not modify any other fields on the mask, particularly the PTS field.

Complete the subroutine Z_ACCESS, i.e. include the name of your subprogram and the respective parameters.

Testing Further Database Operations

Copy an existing push button or modify the push button label.

Modify the operation code PZ_AS_OPERATION in the click event for the push button.

If necessary, you can rename the label of an existing push button.

Creating a User Exit for Single-object Processing

The table administration system contains a number of standard validations. Individual validations can be executed via this user exit.

It can also be used for data conversions.

- Invoking the User Exit
- Creating the User Exit

Invoking the User Exit

The user exit is invoked twice:

- immediately before accessing the table data, and
- immediately after invoking access to the table data.

Creating the User Exit

Use an existing module as a basis.

Save it under a different name.

Warning:
Do not modify the basis module.

The following naming convention is used for access modules:

xxxAS00y

where *xxx* is the object code and *y* is the Natural object type.

Basis	Library	Save as
ZXFCAS0A (PDA)	SYSCOMP	xxxAS00A (as parameter data area)
ZXFCAS1N (Text)	SYSCOMP	xxxAS00N (as subprogram)

Parameter Data Area

Enter the fields for the table using exact formats and lengths.

Warning:
Do not modify the field P_PTS or the length of the key ID fields.

Subprogram

Complete the place holder for the parameter data area, field names, table names, client ID and language code according to the instructions in the subprogram.

Warning:
Do not modify variables beginning with PZ_AS, etc.

The following variables are exceptions and may be modified:

PZ_AS_RSP,
PZ_AS_MSG(*)
PZ_AS_FLD_POS must be set with P_XXX_<field-name>_POS in the case of an error.

The contents of your data fields are in the variables P_xxxx of the parameter data area xxxAS00A.

To control when a job is executed, use the variables LZ_AFTER_ACCESS and LZ_BEFORE_ACCESS.

LZ_BEFORE_ACCESS is used before the table data is accessed.

LZ_AFTER_ACCESS is used after the table data is accessed, immediately before data output.

The following operation codes can be used for job control:

LZ_XA_STORE
LZ_XA_UPDATE
#ZCA_DEL_DESC_i(i=1...4)
#ZCA_READ_BY_DESC_ i(i=1...4)

Note:
Transfer the contents of P_xxxx into the internal format **only** when you have modified data.

Generating Functions in Frame Gallery

The frame gallery provides pretested and error-free frames that you can use to generate functions. Functions are dialogs and other Natural modules. When you use a frame, you need not "create" the dialogs yourself in the dialog editor.

There are two types of frames:

- **Prototype Frames:** This type of frame is used to create a prototype. Database access is not included. It is intended to provide a quick preview of the dialog layout which can then be discussed with, for example, the business department.
  If you want to insert controls such as command buttons or any additional functionality into the generated prototype, you can use the dialog editor.
- **Production Frames:** This type of frame is used to generate dialogs. It is linked with an object view to provide database access.
  An object view consists of access modules, local data areas and parameter data areas which allow database data for a given object type to be created, read, updated and deleted.

The following topics are covered below:

Criteria for Using Frame Gallery Frames

The frame gallery can be used to produce object views, dialogs and other modules, assuming the following prerequisites are met:

- the search key is unique;
- the key length does not exceed 65 bytes;
- the key is not contained in a periodic group and is not a multiple-value field.

Subdescriptors can be used, but generated code will require update if a field is used for more than one subdescriptor in a superdescriptor.

Superdescriptors can only be used if DDMs include descriptions of the structure of the superdescriptor. If you use a superdescriptor containing more than two components, you are advised to update the generated key and browse dialog layouts.

Keys other than alphanumeric, numeric, and packed are not fully supported. If you use such keys, you will need to add MOVE EDITED statements to the generated code.

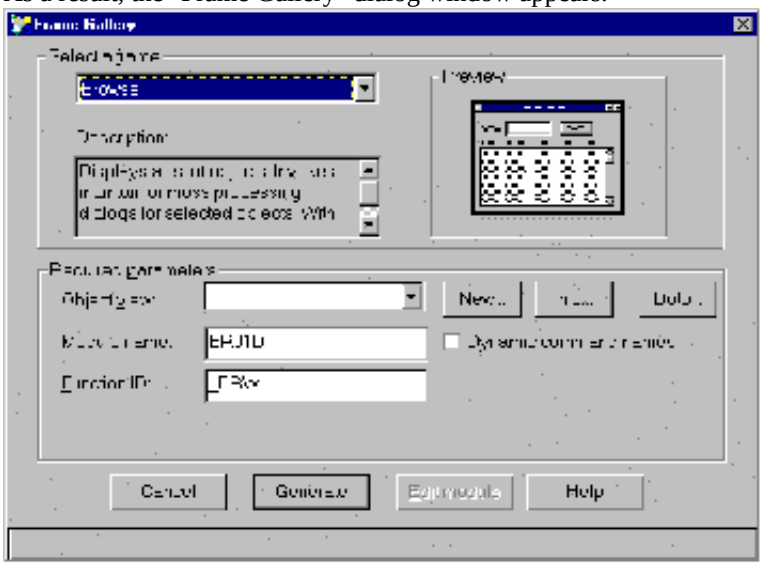
You may wish to improve the display formats of some fields in list boxes and dialog layouts.

Accessing the Frame Gallery

To access the frame gallery

1. Open the database and start Natural.
2. From the Object menu, choose Generate. Or choose the Generate tool-bar button:

As a result, the "Frame Gallery" dialog window appears.



Creating an Object View

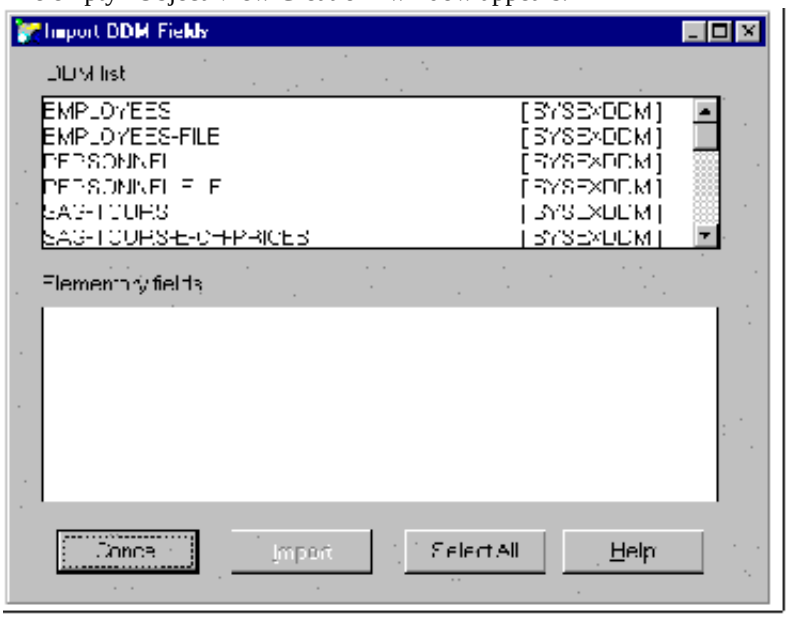
When creating a new dialog, you must define an object view.

An object view is a set of Natural modules used for standard database access (store, update, delete, read, list) for a number of fields selected from a data definition module (DDM).

To create an object view

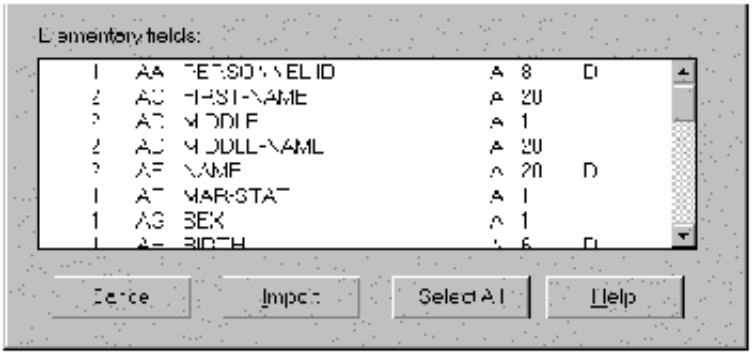
You first define the DDM and select the DDM fields that will be used.

1. In the "Frame Gallery" dialog window, choose the New button.
   The empty "Object View Creation" window appears.

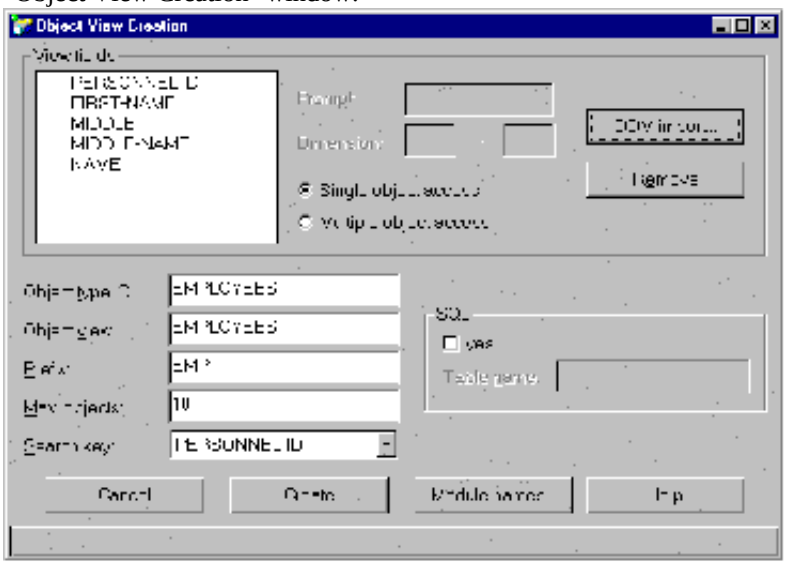


2. In the "DDM list" list box, select the DDM.

All the elementary fields of the DDM you selected are shown in the ’’Elementary fields" list box.



3. In the "Elementary fields" list box, select the fields you want.
   To select more than one item, press CTRL while you click the item.
   You can also select a range of items by selecting the first item and then pressing SHIFT and clicking the last item.
   Or drag the mouse over a range of items.
   To select all items, choose the Select All button.
4. Choose the Import button.
   All selected fields are imported and the window is closed. The imported fields are then shown in the "Object View Creation" window.

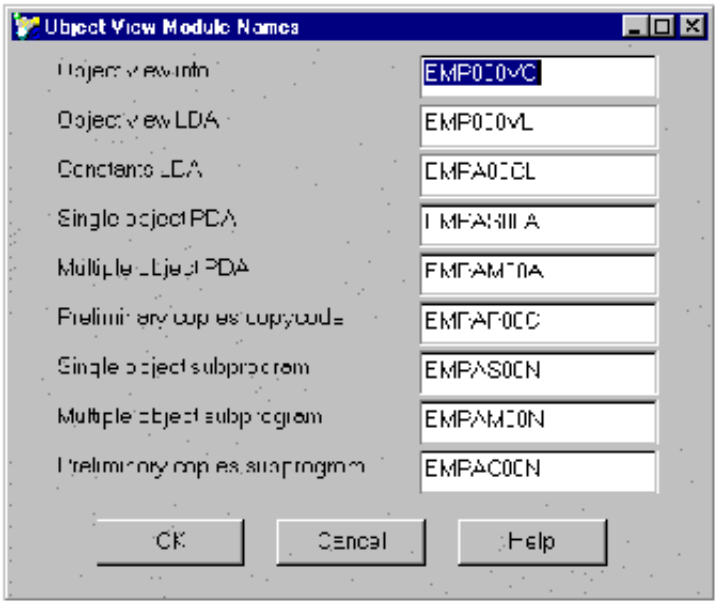


   Optional - You can specify a prompt (label) to be used for a specific input field in the generated dialogs. If you do not specify a prompt, the prompt is derived from the name shown in the "View fields" list box.
5. From the "View fields" list box, select the field.
   The default prompt appears in the "Prompt" text box. You can modify the prompt in this text box.
   Optional - For fields with a dimension, the dimension is set by default to 1. If necessary, you can update the dimension.
6. From the "View fields" list box, select the field.
   The field’s dimension appears in the "Dimension" text box. You can modify the dimension in this text box.
   The following parameters are used during the generation process. They can be modified.

Object type	By default, this is set to the DDM name. It is used to identify the object type in the application shell.
Object view	The logical name of the object view to be used in the generated module.
Prefix	The prefix used for field names, module names, function ID, etc. in the application shell.
Max. objects	This parameter applies to multiple object access: the maximum number of objects that are to be accessed with each call to the multiple object access module.
Search key	Search key for database access. The key must uniquely identify each object.
SQL	When you select this check box, you can create SQL access modules. The SQL table that you specify in the "Table name" text box will then be used. To remove search key components that are not required, select the "Search key" option button. Then select the fields from the "View fields" list box and choose the Remove button. Note that the search key consists of a maximum of five components.
Dynamic command names	By default, generated dialogs use fixed command names. If you check this box, command names are retrieved at runtime from the application shell's command definitions. Using the application shell, you can change dynamic command names without having to update dialogs. You can also translate command names into additional languages.

All fields that you have imported are available both for single-object access (for example in the maintain dialog) and multiple object access (for example in the key selection dialog).
Optional - You can remove fields that are not required for multiple-object access or single object access.

7. Select the "Multiple object access" option button.
   Or select the ''Single object access" option button.
8. Hold down CTRL and select all fields not required.
9. Choose the Remove button.
   Only the required fields are now displayed in the "View fields" list box.
   Optional - You can display and/or modify the names of the Natural modules that are to be generated.
10. To display the names of the modules, choose the Module Names button.
    The "Object View Module Names" window appears.

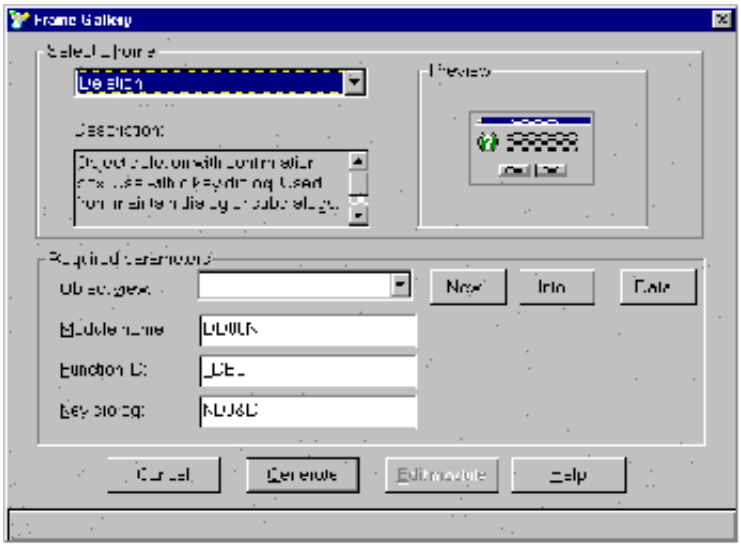


From this window you can rename the modules that will be generated.
When the object view has been created, you will find the generated Natural modules in your Natural library window.

Generating Dialogs

When at least one object view has been defined, you can generate dialogs.

The description below the frame name provides a brief description of the selected frame. It also informs you whether additional dialogs are required.



The "Object view" drop-down list box contains a list of available object views.

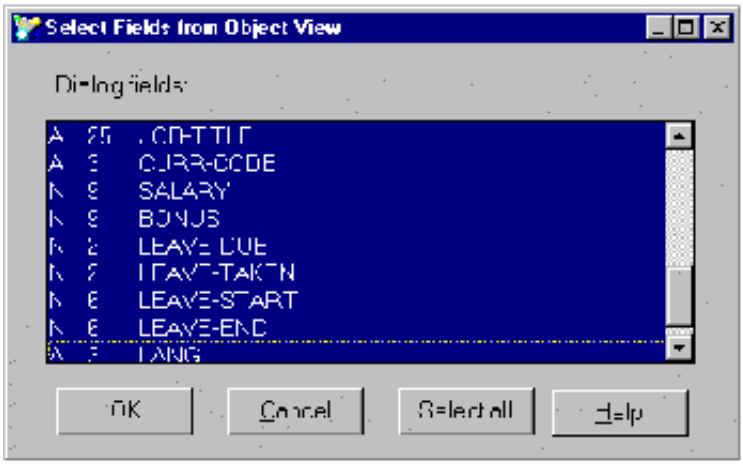
To generate a dialog

1. Optional - To add a new object view, choose the New button.
   The "Object View Creation" window appears.
2. Optional - To display information about the object view, choose the Info button.
   The "Object View Creation" window appears.
   For frames which include a list box or a data entry component, you can select fields to be included in the default dialog layout. By default, the following fields are selected for display.

Data entry dialog component	All fields read by the selected object view's single object access module. With the exception of the nonstandard dialog, the list does not include the search key, which is included as a separate dialog component.
List box component	All fields read by the selected object view's multiple object access module, including the search key.

3. To deselect some of the preselected display fields, choose the Data button.
   The "Select Fields from Object View" window appears.

Function ID	Frame Type
_BRW	Browse dialog
_MNT	Maintain dialog
_DEL	Deletion subprogram
_MASS	Mass processing dialog
_NST	Non-standard dialog



   To select more than one item, you press CTRL while you click the item.
   You can also select a range of items by selecting the first item and then pressing SHIFT and clicking the last item.
   To select all items, you choose the Select All button.

4. Choose the OK button to return to the "Frame Gallery" window.
   The "Module name" text box contains the name of the module that is to be generated. The name depends on the frame you have chosen. This name can be modified.
   The "Function ID" text box contains the ID that is used in the application shell in order to invoke the generated dialog. The ID can be modified. The following IDs are available:

Function ID	Frame Type
_BRW	Browse dialog
_MNT	Maintain dialog
_DEL	Deletion subprogram
_MASS	Mass processing dialog
_NST	Non-standard dialog

   The "Key dialog" text box contains the module name of a key dialog that is required for a maintain dialog or deletion dialog. The key dialog can be generated before or after you generate a delete or maintain dialog. However, the key dialog must exist before the delete or maintain function can be invoked.
   If this is the first dialog for the object type, the "Add Object Type" window appears.
   Additionally, when you generate a browse, maintain, delete, mass processing, or nonstandard dialog, the "Add Function" window appears.
   These windows are similar.

5. In the "Name" text box, enter the name of the object type or function.
6. In the "DIL text" text box, enter the text to be displayed in the Dynamic Information Line for this object type or function.
7. Choose the OK button.
   The dialog is created.
   Optional - Choose the Edit module button.
   The newly created dialog is loaded into the dialog editor and you automatically leave the frame gallery.

Customizing a Generated Application

Once the dialogs and other modules have been generated using the frame gallery, you use the Natural dialog editor to modify the dialog layout and to customize the generated code to conform to application-specific requirements.

Modules generated with the frame gallery include:

- Internal frame logic. The internal frame logic, or frame code, covers a large amount of the processing required to create a working module.
- Application specific code. Based on the DDM and fields you select, appropriate code is generated to define data structures, access and update database information, and produce dialog layout. This generated code meets most simple requirements, although you will need to modify the generated code to handle binary data, keys with unusual data types, or to improve dialog layouts.
- Customizable components which contain generated code and suggested code. Suggested code is included as a Natural comment. Suggested code helps you add further applications-specific code for your own requirements, for example to: link dialogs, handle validation, or provide active help. The customizable components can be modified and/or extended.

In modules other than dialogs, the components above are simply included together and are all modifiable using the appropriate editor.

In dialogs, the internal frame logic is included in protected code segments which are not visible in the dialog editor. The protected code segments make extensive use of copycode.

The following topics are covered below:

- Customizable Components
- Generated Code
- Suggested Code
- Integrating a Dialog in the Application Shell

Customizable Components

The generated application-specific code and the suggested code in dialogs are included in customizable components. These are inline subroutines which are visible in the dialog editor, and which you can modify to meet your own requirements. Help is available in the dialog editor to explain how each customizable component can be used. The section Application Frames explains which customizable components are used by each frame. The section Customizable Components explains each customizable component in detail.

When you add application-specific code or modify the generated code, it is important to understand how the internal frame logic works and how you can control its behavior.

The following topics are covered below:

- Commands
- Frame Logic Control Variables
- Skeleton Objects

Commands

The processing in dialogs is triggered by events. These events are handled initially by the internal frame logic which translates them into commands. The standard commands are described in section Standard Commands.

The frame control logic calls different customizable components depending on the command being processed. See section Application Frames for a list of the components activated when a command is processed and the sequence in which these components are activated.

The frame control logic is controlled by numerous variables, which have a default setting, as described in section Application Frames. By changing these variables, it is possible to modify the internal behavior of the frame control logic contained in the protected code segments of each frame.

Frame Logic Control Variables

You can influence how the internal frame logic works by setting frame logic control variables. Some of these variables are set in the generated code included in customizable components and you can change their settings. Additional variables have defaults and are not explicitly set in the generated code. However you can add code to change the default (usually in the initialize subroutine). See section Application Frames for a list of variables, their usage and default settings.

Reusable Components

When you customize modules produced using the frame gallery, you can call various reusable components from the generated code. The reusable components are subroutines, subprograms, and other objects). These components are described in the section Reusable Components.

Skeleton Objects

Additional skeleton objects are provided to create some advanced components, for example in list box processing.

To implement a module

1. Copy the skeleton object into a new object.
   Do **not** modify the skeleton itself.
2. Fill in the suggested code (data definition and coding) for the new object.

Generated Code

Prototype frames contain virtually no generated code.

Production frames include the basic processing required for simple data maintenance functions based on the DDM and fields selected by the user when creating the object view and dialog. It includes:

- a basic dialog layout;
- inclusion of references to the LDAs, PDAs and subroutines associated with the selected object view;
- logic to handle database access and transfer of data between the user interface and the database; and
- logic handling some standard navigation links between dialogs (for example, the linking of a key dialog with a maintain dialog).

This generated code is adequate for very simple functions. It is normally necessary to make changes to the generated dialog for the following reasons:

- to improve the dialog layout;
- to add code to handle requirements for data validation, improve handling of numeric fields, etc.; and
- to link dialogs, for instance to link a maintain dialog with an associated subdialog.

Suggested Code

The suggested code can be regarded as providing a model for the implementation of functions of medium complexity.

Suggested code statements are contained in the customizable components as comment lines. They consist of syntactically correct Natural statements. Values which require customizing, for example context-specific variables, are in lower case.

- Naming Conventions in the Suggested Code
- Skeleton Data Definitions

Naming Conventions in the Suggested Code

To assist in the use of SCAN/REPLACE in suggested code, certain conventions have been followed:

Placeholder	Description
view	indicates view names
xxx	indicates a view or area prefix
#xxx	prefix for user-defined variables

There are also other placeholders that you must modify manually. For example, "field1" must be replaced by a field name.

Depending on program-specific requirements, you can also add new code.

Skeleton Data Definitions

In prototype frames, there is suggested code for data definitions, for example placeholder local data areas or parameter data areas for view fields that are present in the form:

xxxAS00A /* Single-object PDA

You must replace the placeholder "xxx" with the appropriate view prefix.

Integrating a Dialog in the Application Shell

If you are using the frame gallery, dialogs are automatically integrated into the application shell. In some cases, however, you must manually integrate the dialog into the application shell.

To integrate a dialog into the application shell

1. Define a new object type (command Object type, action New) in the application shell.
2. Define a new function (command Function, action New) in the application shell.
   Depending on the dialog type you create, we suggest you use the following function IDs, where "xxx" is the prefix you defined for the object view:
   - xxx_BRW (browse dialog)
   - xxx_MNT (maintain dialog)
   - xxx_DEL (delete subprogram)
   - xxx_MASS (mass processing dialog)
   - xxx_NST (nonstandard dialog)
3. Select the menu command Options and choose Refresh Initialized Data.
   Once the function is defined, it can be called via the ’’Direct Call" dialog.
   To integrate the new function into the graphical navigation, see the Natural Application Shell documentation.

Communication Between Dialogs

Event-driven applications require communication between various dialogs. New dialogs must be opened, and information must be exchanged between existing dialogs.

This section describes the various methods for communication between dialogs.

- The Standard Interface
- Calling a Dialog

The Standard Interface

All dialogs implemented using frames have a standard interface. Parameters are defined in the parameter data area ZXXREC0A. All variables of this parameter data area are collected within the group PZ_RECEIVE.

Standard Interface Structure

The standard interface is divided into two areas, PZ_STRUCTURE and PZ_DATA.

The group PZ_STRUCTURE contains all variables related to application control. These variables are used primarily by the frames.

The array PZ_DATA (A100/1: 40) is used for the transfer of technical data between dialogs. For example, an array #PZ_DATA (A100/1:n) can be defined and then redefined according to specific requirements. PZ_DATA is not used by the frames.

Example:

```
01 #PZ_DATA (A100/1:2)
01 REDEFINE #PZ_DATA
02 #COMMENT (A250)
02 #NAME    (A30)
.
.
   MOVE '   ' TO #COMMENT
   MOVE '   ' TO #NAME
   MOVE #PZ_DATA (1:2) to PZ_DATA (1:2)
```

Local Copy of the Interface

The parameter values are no longer available following the processing of the EVENT or OPEN DIALOG statements to which they are passed. Therefore, those parameter values which are required for a longer period of time must be saved as local variables. A local copy of the parameter data area (Natural object ZXXLOC0A) is available for this purpose. These variables are collected in group PZ_LOCAL.

During execution of internal commands, the frames transfer the relevant parameters from the group PZ_RECEIVE into the group PZ_LOCAL.

Communication Using User-Defined Events

User-defined events can be added in the dialog editor for communication between dialogs (see the *Natural User's Guide*). This is necessary if customized processes are to be executed.

Communication using Pre-Defined Event Z_CMD_EXEC

Using the pre-defined event Z_CMD_EXEC, it is possible for another dialog to process certain commands. The following suggested code can be used to do so (appropriate modifications must be made based on specific requirements):

```
MOVE ....     ..........TO PZ_LOCAL....
MOVE 'command'           TO PZ_LOCAL.PZ_CMD_ID
MOVE LZ_CMD_TYPE_INT     TO PZ_LOCAL.PZ_CMD_TYPE
SEND EVENT 'Z_CMD_EXEC' TO dialog-id
 WITH PZ_LOCAL
```

You can send any information via group PZ_LOCAL which contains PZ_DATA.

Calling a Dialog

A dialog call can be performed in various ways. The first dialog call occurs using an OPEN DIALOG statement. The communication between existing dialogs is accomplished using the SEND EVENT statement.

The frames perform for the most part the communication between dialogs. There are, however, situations in which communication must be implemented within customizable components. For example:

- Browsers and entry dialogs for functions can be started using the corresponding commands. Subdialogs can also be opened using commands.
- The opening of modal windows and key dialogs for selection of foreign keys must be individually coded.
- The exchange of information between dialogs.

Communication with Subdialogs

The subroutine Z_ASSIGN_SUBDIALOG in the maintain dialog is used to specify any subdialogs the maintain dialog calls. You must modify the suggested code to specify the module name of each subdialog and the local command ID you are associating with it and the total number of subdialogs associated with the maintain dialog. You must also associate the local command with a push button and define the command ID you specify in the application shell.

The actual communication between a subdialog and a maintain dialog is carried out by the internal frame logic. When the local command is invoked, either the corresponding dialog is opened, or, if it is already open, it will receive the focus.

After the subdialog has been opened, there can be further communication between the subdialog and the maintain dialog.

The frame logic for the subdialog informs the maintain dialog whenever data contained within the subdialog has been modified.

The frame logic of the subdialog or maintain dialog advises of the execution of commands which have an impact on the other dialog. The frame descriptions indicate which commands cause this type of communication.

Controlling a Subdialog from another Subdialog

If a subdialog is to be controlled from another subdialog, the command assigned in the maintain dialog must be passed from the subdialog to the maintain dialog. For this purpose, the following suggested code must be added and adapted to the subroutine Z_CUSTOM_CMD in the subdialog:


```
IF LZ_STD.LZ_FRAME_CMD_ID EQ 'command'
  MOVE LZ_STD.LZ_FRAME_CMD_ID    TO  PZ_LOCAL.PZ_CMD_ID
  MOVE LZ_STD.LZ_FRAME_CMD_TYPE  TO  PZ_LOCAL.PZ_CMD_TYPE
  SEND EVENT 'Z_CMD_EXEC'        TO  PZ_LOCAL.PZ_MAIN_DLG
   WITH PZ_LOCAL
END_IF
```

Foreign Key Selection/Active Help

The key dialog for input/output of primary keys is opened by the frame of the maintain dialog. This is done during the processing of standard commands whenever the input of a key value is required.

If a key dialog is opened to select a foreign key, the following suggested code can be added and adapted:

```
MOVE LZ_KEY_TYPE_FOREIGN  TO  LZ_STD.LZ_KEY_TYPE
MOVE #field               TO  PZ_LOCAL.PZ_SEL_KEY

OPEN DIALOG 'xxxKD0&D'  #DLG$WINDOW WITH PZ_LOCAL
```

When a key value is selected, the component Z_RECEIVE_KEY is executed in the calling dialog and the key value can be transferred to the corresponding field.

Note:
If from one dialog, selection help for various fields is to be called, the field for which selection help is to be opened must be marked with a user-defined indicator. During the transfer of the key value in component Z_RECEIVE_KEY, the corresponding field must be passed based on the identifier.

Calling Modal Windows

Modal windows are not opened by frames. The open dialog can be coded either in an event handler or in a specialized component. The following suggested code can be added and adapted:

```
MOVE data               TO  PZ_LOCAL.PZ_DATA(1:n)

OPEN DIALOG 'xxxMW0&D#  #DLG$WINDOW WITH PZ_LOCAL
```

Following the modification confirmation in the modal window, the component Z_RECEIVE_DATA is executed in the output dialog. The modified data from the array PZ_RECEIVE.PZ_DATA can then be transferred to the corresponding local variables.

Note:
If from one dialog, multiple modal windows are to be called, the window to be opened must be marked with a user-defined indicator. This indicator is also relevant during data transfer.

Commands for Opening a Dialog

The opening of certain dialogs can be invoked using a special type of command as defined in the *Natural Application ShellManual*. The frames of the dialog in which the commands are to be invoked open automatically the desired dialog. They need not be programmed.

Starting an Application

A dialog for icon-based navigation can be started with the command type "Start an application".

Starting a Browser

Browse functions can be started using the command type "Start a browser".

Starting a Function

Functions can be started using the command type "Start a function". A key is provided using the variable PZ_LOCAL.PZ_SEL_KEY. This is used for the function start if the switch PZ_LOCAL.PZ_KEY_FILLED is set to TRUE.

In maintain dialogs, a new function with the same Action and Object Type is started using the command Z_OPEN.

In maintain dialogs, for the action type New, a new function with the same object type and the selected action is started.

For browse dialogs, for the commands of type Action, a function for each selected data record is started.

For a description of how to start applications, browse functions and other functions, see Starting a Dialog (Application, Function, Browse).

Application Frames

The following topics are covered below:

- Frame Overview
- Browse Dialog
- Deletion Subprogram
- Key Dialog
- Maintain Dialog
- Mass Processing Dialog
- Modal Window
- Nonstandard Dialog
- Subdialog
- Background Program
- Load Objects Subprogram
- Unload Objects Sub

Frame Overview

Frame	Description
Browse dialog	Displays a list of objects. The maintain or mass processing dialog or the deletion subprogram can be invoked for selected objects.
Deletion subprogram	Deletes an object, displaying a message box to request confirmation of the deletion from the user. It requires an associated key dialog.
Key dialog	Used to provide open object, save as and active help windows. It allows the input of an object ID or the selection of an object from a list.
Maintain dialog	Allows the creation, display, and update of individual objects. It requires an associated key dialog.
Mass processing dialog	Allows the creation, display, update and deletion of multiple objects. This dialog does not use preliminary copies.
Modal window	Multi-purpose modal window. No default dialog layout/data access is produced. The dialog includes standard navigation and command handling logic.
Nonstandard dialog	Includes standard logic for navigation and command handling.
Subdialog	Used in association with a maintain dialog. (The link must be manually coded.)
Background program	This program is used to implement background programs which start from an online program using subprogram ZXBG010N.
Load objects subprogram	This subprogram is used to load objects of a specified type into the database from a workfile.
Unload objects subprogram	This subprogram is used to unload objects of a specified type from the database to a workfile.

For each dialog type except for modal windows, there is both a production and prototype variant of the program frame.

Production versions of each dialog type (except modal windows) include data access and a default dialog layout for the accessed data.

Prototype versions do not include data access or a default layout for the data accessed.

Browse Dialog

Description

Search and overview functions are provided by this frame.

As result of a search, data records are displayed in a list box. By selecting data records and choosing an action, the corresponding function is called for each of the selected data records.

This frame distinguishes between the mass processing action and all other actions.

For the mass processing action, the function is only called once for all selected data records. The mass processing and browse dialogs then communicate with one another using commands.

For all other actions, a separate function is started for each selected data record.

Links with Other Dialogs

Called:	Directly through the command system
Calls:	Maintain dialog Deletion subprogram Mass Processing dialog

Dialog Layout

Component	Comments
Search key	Up to 5 search key components. If the key includes more than 2 components, it is advisable to improve the layout using the dialog editor.
List box	Displays fields read using the multiple object access module.
Push buttons	Search

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_ACCESS_DATA	X	
Z_ASSIGN_INPUT_TO_KEY		
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		
Z_FILL_ITEM		X
Z_INITIALIZE		
Z_PASS_KEY		X
Z_PROCESS_ITEM		X
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_SET_KEY_RANGE		X
Z_UPDATE_ITEM		X

Commands Supported

Command	Explanation	Components Activated
Z_LB_CLICK	Selection in list box changed	
Z_LB_FILL	Fill-event in list box occurred	Z_ACCESS_DATA Z_FILL_ITEM
Z_LB_SELECT	Process reaction on change of selection in list box	
Z_CANCEL_DLG	Mass processing gives notice of ending	
Z_CANCEL_KEY	Key dialog gives notice of ending	
Z_CANCEL_TMR	Cancel timer activated after runtime error	
Z_CLOSE	Close dialog	
Z_EXIT	End application Comments: A timer is activated that triggers the command Z_EXIT_TMR.	
Z_EXIT_TMR	Close dialog due to ending of application	
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key	Z_RECEIVE_KEY
Z_ITEM_ADD	Another dialog sends new data records for insertion in the list box	Z_UPDATE_ITEM Z_PASS_KEY
Z_ITEM_DEL	Another dialog sends keys for deletion from the list box	Z_UPDATE_ITEM Z_PASS_KEY
Z_ITEM_MOD	Another dialog sends data records for modification in the list box	Z_UPDATE_ITEM Z_PASS_KEY
Z_ITEM_NEXT	Mass processing expects key IDs of the next data record selected	Z_PASS_KEY Z_SELECT
Z_ITEM_PREV	Mass processing expects key IDs of the data record previously selected	Z_PASS_KEY Z_SELECT
Z_SELECT_ALL	Select all list box entries	
Z_SEARCH	Search is started anew Z_ACCESS_DATA Z_FILL_ITEM Z_PASS_KEY	Z_SET_KEY_RANGE
Z_START_KEY	Start with key	Z_INITIALIZE
Z_START_NKEY	Start without key	Z_INITIALIZE
Type of command Action	Function start for marked data records (1) Only when list box is empty (2) Only when list box is not empty	(1)Z_ASSIGN_INPUT_TO_KEY (2) Z_PASS_KEY (2) Z_SELECT

Associated Variables

Name	Value	Comment
LZ_ALIGN_HORIZ	**True**	When changing the dialog size, the list box size is adjusted vertically.
	False	No vertical adjustment of the list box size.
LZ_ALIGN_VERT	**True**	When changing the dialog size, the list box size is adjusted horizontally.

Name	Value	Comment
	False	No horizontal adjustment of the list box size.
LZ_ADD_EVERY	True	Newly added data records are included in the list box, independent of whether they fit in thecurrent range of records in the list box.
	False	Only newly added data records that fit in the current range of records in the list box are included in it.
LZ_ADD_ON_EMPTY	**True**	If the list box is empty, each newly added data record is included in the list box.
	False	Newly added data records are only included in an empty list box if there are no data (EOD).
LZ_BOX		Group consisting of LZ_BOX_ITEM and LZ_BOX_ITEM_KEY. The group can be used for MOVE BY NAME statements as a simple way of transferring fields in or out of database fields.
LZ_BOX_ITEM		Matches the attribute STRING of a list box item. If several columns are displayed in the list box, this variable must be redefined accordingly.
LZ_BOX_ITEM_KEY		If the key is not completely contained in the list box, the missing parts must be defined here.
LZ_DOUBLE.LZ_CMD_ID		Sequence of commands (actions) for double-click on a list box item. The first action allowed in the table counts as a command that is executed with a double-click.
LZ_START_USER_INPUT	True	If the list box is empty or no entry selected and an action (not type add) chosen, the subroutine Z_GET_KEY_USER_INPUT is called and then the function is started with the key LZ_SELECT_KEY.
	False	No reaction when an action is chosen and the list box is empty or no entry is selected.
LZ_KEY_IN_LISTBOX	**True**	The list box contains the entire key of a data record.
	False	The key of a data record is **not** completely contained in the list box.
LZ_MARGIN_RIGHT	**40**	Right margin in pixels between dialog and list box (only when LZ_ALIGN_HORIZ = TRUE).
LZ_MARGIN_BOTTOM	**40**	Bottom margin in pixels between dialog and list box (only when LZ_ALIGN_VERT =TRUE).
LZ_REC_EOD	True	No further data present.
	False	Further data present.
LZ_REC_FOUND		Number of data records that are found with an access.
LZ_REC_IND		Index of the current data record in the table of a multiple-record access module.
LZ_REC_NUM_FILL FILL	**10**	Number of data records to be read following an event on list box.
LZ_REC_NUM_SEARCH	**0**	Number of data records that are read for command **Z_SEARCH**. Depending on the current screen resolution, the variable is set by the frame to a multiple of LZ_REC_NUM_FILL.
LZ_SELECT_KEY		Key value for the current data record. If the key consists of several components, LZ_SELECT_KEY is to be redefined accordingly.

Name	Value	Comment
LZ_START_SEARCH	**True**	The search begins directly after the dialog is started. (Default for production frame)
	False	The start of the search must be explicitly triggered by the user. (Default for prototype frame)

Variables for Controlling Frame Behavior

The following variables for controlling frame behavior do not appear in the suggested code. Values deviating from the default can be assigned in the customizable component Z_INITIALIZE.

- LZ_ALIGN_VERT,
- LZ_ALIGN_HORIZ,
- LZ_START_USER_INPUT,
- LZ_KEY_IN_LISTBOX,,
- LZ_MARGIN_RIGHT,
- LZ_MARGIN_BOTTOM,
- LZ_REC_NUM_FILL,
- LZ_REC_NUM_SEARCH,
- LZ_ADD_EVERY,
- LZ_ADD_ON_EMPTY

Deletion Subprogram

Description

Deletes an object, displaying a message box to request confirmation of the deletion from the user. It requires an associated key dialog.

Links with Other Dialogs

Called:	Directly through the command system from browse, maintain and mass processing dialogs.
Calls:	None

Dialog Layout

Standard deletion confirmation box.

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_DELETE	X	
Z_LOCK_RECORD		X
Z_INITIALIZE		X

Associated Variables

Name	Value	Comment
LZ_DELETE_KEY		Record key to be deleted. This variable is to be redefined to correspond to the data transferred.
PZ_MSG.PZ_MSG_FILL (1)		This variable should be set to the value of the key is to be used in display format.

Key Dialog

Description

Used to provide open object, save as and active help windows. It allows the input of an object ID or the selection of an object from a list.

Links with Other Dialogs

Called:	To select object (maintain dialog, deletion subprogram) to specify key of new object (maintain dialog, mass processing dialog) to specify key of object SAVE AS (maintain dialog)
Calls:	None

Dialog Layout

Component	Comments
Search key	Up to 5 search key components. If the key includes more than 2 components, it is advisable to improve the layout using the dialog editor.
List box	Displays fields read using the multiple object access module.
Push buttons	OK, Cancel, Search

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_ACCESS_DATA	X	
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		
Z_FILL_ITEM		X
Z_INITIALIZE		X
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_RETURN_KEY		X
Z_SELECT		X
Z_SET_KEY_RANGE		

Available Commands

Command	Explanation	Components Activated
Z_LB_CLICK	Selection in list box changed	
Z_LB_DOUBLE occurred	Double-click in list box Z_RETURN_KEY	Z_SELECT
Z_LB_FILL	Fill event of list box occurred Z_FILL_ITEM	Z_ACCESS_DATA
Z_LB_SELECT	Return selected list box item	Z_SELECT
Z_CANCEL	Close dialog	
Z_CANCEL_KEY	Key input was interrupted	
Z_CLOSE	Close dialog	
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key Comments: (1) Only when previous command was Z_READ. (2) For change of action within a dialog, additionally the same components as under 2. are run through.	1. Selection of a foreign key: Z_RECEIVE_KEY 2. Selection of a key at the start of the processing: Z_INITIALIZE (1) Z_CHECK_EXISTENCE Z_LOCK RECORD Z_ADD_PREL_REC Z_FILL_DIALOG 3. Input of a key at the close of the processing: Z_CHECK_EXISTENCE Z_LOCK RECORD Z_UPDATE_PREL_KEY Z_VALIDATE Z_UPDATE_PREL_REC Z_ACTIVATE_PREL_REC (2)
Z_OK	Pass value to calling dialog Close dialog	Z_RETURN_KEY
Z_SEARCH	Start of a search	Z_SET_KEY_RANGE Z_ACCESS_DATA Z_FILL_ITEM
Z_START_SEL	Start as selection dialog	Z_INITIALIZE
Z_START_SAVE	Start as storage dialog	Z_INITIALIZE

Associated Variables

Name	Value	Comment
LZ_BOX		Group consisting of LZ_BOX_ITEM and LZ_BOX_ITEM_KEY. The group can be used for MOVE BY NAME statements as a simple way of transferring fields in or out of database fields.
LZ_BOX_ITEM		Matches the attribute STRING of a list box item. If several columns are displayed in the list box, this variable must be redefined accordingly.
LZ_BOX_ITEM_KEY		If the key is not completely contained in the list box, the missing parts must be defined here.
LZ_KEY_IN_LISTBOX	**True**	The list box completely contains the key of a data record.
	False	The key of a data record is **not** completely contained in the list box.
LZ_REC_NUM_FILL	**10**	Number of data records to be read following a FILL event on list box.
LZ_REC_NUM_SEARCH	**0**	Number of data records that are read for command **Z_SEARCH**. Depending on the size of the list box, the variable is set by the frame to a multiple of LZ_REC_NUM_FILL.
LZ_SELECT		Group to enable transfer of input fields using MOVE BY NAME into other fields. Contains as only field LZ_SELECT_KEY
LZ_SELECT_KEY		This variable must be redefined to correspond to the key to be transferred.
LZ_START_SEARCH	**True**	The search begins directly after dialog start. (Default for production Frame)
	False	The start of the search must be explicitly triggered by the user. (Default for prototype Frame)

Variables for Controlling Frame Behavior

The following variables for controlling frame behavior do not appear in the suggested code. Values deviating from the default are to be assigned in the customizable component Z_INITIALIZE.

- LZ_KEY_IN_LISTBOX,
- LZ_REC_NUM_FILL,
- LZ_REC_NUM_SEARCH

Maintain Dialog

Description

Maintenance functionality is implemented in this frame. A dialog of this type implements "new", "modify" and "display" actions.

If not all attributes of the object to be processed can be presented in one dialog, these can be distributed among further dialogs. These dialogs should be implemented using the frame for subdialogs or for modal windows. The maintain dialog communicates with subdialogs.

Modifications of data are made to preliminary copies of the data records. The original data are only modified when a Save command is issued.

Links with Other Dialogs

Called:	Directly through the command system from a browse dialog
Calls:	Key dialog (required) Deletion subprogram Subdialogs (handcoded link)

Dialog Layout

Component	Comments
Data entry	Allows update of fields read using the single object access module, excluding the search key.
Push buttons	Close, confirm

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_ACTIVATE_PREL_REC	X	
Z_ADD_PREL_REC		X
Z_ASSIGN_DEFAULT_KEY		X
Z_ASSIGN_SUBDIALOG		
Z_CHECK_EXISTENCE		X
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		
Z_FILL_DIALOG		X
Z_INITIALIZE	X	X
Z_LOCK_RECORD		X
Z_NAVIGATE_ON_ERROR		
Z_READ_PREL_REC		X
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_UPDATE_PREL_KEY		X
Z_UPDATE_PREL_REC		X
Z_VALIDATE		

The following standard commands are available:

- Z_APPLSTART,
- Z_CALL,
- Z_GET_GLOBAL,
- Z_HELP,
- Z_HELPCONTNT,
- Z_HELPUSE,
- Z_HELPSEARCH

Available Commands

Command	Explanation	Components Activated
Z_CANCEL	Interrupt processing	
Z_CANCEL_DLG	Subdialog gives notice of closing	
Z_CANCEL_KEY	Key input was interrupted	
Z_CANCEL_TMR	Cancel timer activated after runtime error	
Z_CLOSE	Close dialog Comments: (1) The same component as for the command Z_SAVE.	(1)
Z_CONF_DLG	Subdialog was confirmed	
Z_CONFIRM	Update preliminary files Z_UPDATE_PREL_REC	Z_VALIDATE
Z_DATA_MOD	Data has been modified	
Z_EXIT	End application Comments: A timer is activated that triggers the command Z_EXIT_TMR.	
Z_EXIT_TMR	Close dialog due to ending of application Comments: (1) The same component as for the command Z_SAVE.	(1)
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key Comments: (1) Only when previous command was Z_READ. (2) For change of action within a dialog, additionally the same components as under 2. are run through.	1. Selection of a foreign key: Z_RECEIVE_KEY 2. Selection of a key at the start of the processing: Z_INITIALIZE (1) Z_CHECK_EXISTENCE Z_LOCK RECORD Z_ADD_PREL_REC Z_FILL_DIALOG 3. Input of a key at the close of the processing: Z_CHECK_EXISTENCE Z_LOCK RECORD Z_UPDATE_PREL_KEY Z_VALIDATE Z_UPDATE_PREL_REC Z_ACTIVATE_PREL_REC (2)
Z_MOD_DLG	Subdialog was modified	
Z_OPEN	Start new processing	
Z_READ	Read in new data record Comments: (1) The same component as for the command Z_SAVE.	(1)

Command	Explanation	Components Activated
Z_REFRESH	Reverse changes since last update of the preliminary copies	Z_READ_PREL_REC Z_FILL_DIALOG
Z_RESET_DLG	Subdialog was refreshed	
Z_SAVE	Store modifications in original data Comment: Only for update functions. For insert with key definition, the key dialog is opened instead.	Z_VALIDATE (1) Z_UPDATE_PREL_REC (1) Z_ACTIVATE_PREL_REC (1)
Z_SAVEAS	Store data under another key with the original data Comment: The key dialog is opened.	
Z_START_KEY	Start with key Z_ASSIGN_CONTROL Z_ASSIGN_SUBDIALOG Comments: (1) Only for update functions. If the key can not be processed, the key dialog is opened.	Z_INITIALIZE Z_CHECK_EXISTENCE Z_LOCK_RECORD (1) Z_ADD_PREL_REC Z_FILL_DIALOG
Z_START_NKEY	Start without key Comments: (1) Only for insert with key definition. Otherwise, the key dialog is opened.	Z_INITIALIZE Z_ASSIGN_SUBDIALOG Z_ASSIGN_DEFAULT_KEY (1) Z_ADD_PREL_REC (1) Z_FILL_DIALOG (1)
Local command allocated to subdialog	Open subdialog or bring into the foreground	
Command of type action with subtype display or modify	Change of action Z_INITIALIZE (2) Comments: (1) The same components as for the command Z_SAVE. (2) Only when switching between actions within the same dialog. (3) Only for update functions	(1) Z_CHECK_EXISTENCE Z_LOCK_RECORD (2) (3) Z_ADD_PREL_REC (2)

Associated Variables

Name	Value	Comment
LZ_USE_DEFAULT_KEY	**True**	When inserting, an artificial key is used until the first store.
	False	When inserting, before processing, the key dialog for input of a key is displayed.
LZ_MAIN_CONFIRM_ALL	**True**	When confirming the input in the maintain dialog, the input of the subdialogs is also confirmed.
LZ_CHECK_MODIFY	**True**	The frame controls the activation of dialog elements allocated to commands using the CHANGE- or CLICK-EVENTS of the dialog elements.
LZ_GEN_TITLE	**True**	The dialog title is generated by the frame.
LZ_DLG_TYPE		The dialog type allocated to the dialog.
LZ_SUB_DLG_CMD_ID(*)		Commands which cause a subdialog to be started.
LZ_SUB_DLG_NAME(*)		Natural names of the subdialogs that are available for using commands
LZ_SUB_DLG_MAX		Number of subdialogs available for using commands
LZ_KEY_DLG_NAME		Natural name of the key dialog
LZ_ACTIVATE_MODULE		Natural name of the activation module
PZ_SEL_KEY		Key passed by key dialog
PZ_KEY		Technical key
PZ_OBJ_ID		Object type of the object to be processed
LZ_LOCK_OBJ_ID		Object type of the data record to be locked
LZ_LOCK_KEY		Key of the data record to be locked
LZ_VAL_ERR	True	During processing, an error has occurred
LZ_FOCUS		Handling of the dialog element to be focussed
PZ_MSG.PZ_MSG_FILL (*)		Additional information for message
PZ_MSG.PZ_MSG_NUM (*)		SYSERR error number for message
PZ_ERR_FLD_POS		Identification of the erroneous field in the interface of the access module concerned
LZ_SUB_DLG_CMD_ID_SEL		Command to start a subdialog that contains an erroneous field
PZ_CMD_ID		Command
PZ_CMD_TYPE_MAIN		Command type
PZ_ACT_TYPE_CUR		Current Action type
PZ_DLG_ID		Natural dialog ID
LZ_KEY_NEW		New key when storing under a new key
LZ_PTS_NEW		New processing time stamp when storing under a new key
LZ_PREL_KEY		Key for access to preliminary data record
LZ_FRAME_CMD_ID		Command to be processed by the frame

Variables Controlling Frame Behavior

The following variables which control frame behavior do not appear in the suggested code. Values deviating from the default are to be assigned in the customizable component Z_INITIALIZE.

- LZ_CHECK_MODIFY,
- LZ_USE_DEFAULT_KEY,
- LZ_GEN_TITLE,
- LZ_MAIN_CONFIRM_ALL

Mass Processing Dialog

Description

A function for mass processing data can be produced either as a combination of a browse dialog with a mass processing window or as a stand-alone mass processing window. The latter can be started directly from the function list. All processing takes place in the quick-input window.

The mass processing function is usually called from a browse dialog: here, the processing of a single object can be triggered by a double-click. With multiple selection of objects, the processing is triggered using the menu bar. As input, the function receives the (first) key value selected in the browse dialog. The data can then be modified and stored. The modifications are transferred into the browse dialog, in so far as the data concerned are visible there. In addition to these modification functions, the following actions are possible:

- reading a new object,
- deleting an object,
- saving a new object,
- requesting the next object (for multiple selection),
- requesting the previous object (for multiple selection), and
- requesting a new object to be processed by double-clicking in the list in the browse dialog.

Data-modifying actions (record, modify, delete) are transferred into the database direct if save is chosen.

Links with Other Dialogs

Called:	Directly through the command system from a browse dialog.
Calls:	None

Dialog Layout

Component	Comments
Data entry	Allows update of fields read using the single object access module, excluding the search key.
Push buttons	Close, Save, New, Read, Next, Previous, Help

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_CHECK_EXISTENCE	X	
Z_CLEAR_INPUT_FIELDS		X
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		
Z_FILL_DIALOG		X
Z_INITIALIZE		X
Z_LOCK_RECORD		X
Z_NAVIGATE_ON_ERROR		X
Z_PASS_KEY		X
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_UPDATE		X
Z_VALIDATE		

Available Commands

Command	Explanation	Components Activated
Z_CANCEL_DLG	Ending of the browse dialog Comment: (1) Components of Z_SAVE	(1)
Z_CANCEL_KEY	Key input was interrupted	
Z_CLEAR	New Comment: (1) Components of Z_SAVE	(1) Z_CLEAR_INPUT_FIELDS
Z_CLOSE	Close Comment: (1) Components of Z_SAVE	(1)
Z_DATA_MOD	Data has been modified	
Z_EXIT_TMR	Close dialog due to termination of application	(1)
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key	Z_RECEIVE_KEY
Z_KEY_MOD	Key value has been modified	
Z_LIST_MOD	New selection from search dialog Comment: (1) Components of Z_SAVE (2) Components of Z_START_KEY	(1) (2)
Z_NEXT	Request next record from the selection list Comment: (1) Components of Z_SAVE Afterwards the next key from the selection list is requested from the browse dialog (2) Components of Z_START_KEY	(1) (2)
Z_NEW_REC	New record from browse dialog Comment: (1) Components of Z_SAVE (2) Components of Z_START_KEY	(1) (2)
Z_PREVIOUS	Request previous record from the selection list Comment: (1) Components of Z_SAVE Afterwards the previous key from the selection quantity is requested from the browse dialog (2) Components of Z_START_KEY	(1) (2)
Z_READ	Read	Z_PASS_KEY Z_CHECK_EXISTENCE Z_LOCK_RECORD Z_FILL_DIALOG
Z_SAVE	Save Comment: (1) Only for new record (2) Only in case of error	Z_PASS_KEY (1) Z_CHECK_EXISTENCE (1) Z_LOCK_RECORD (1) Z_VALIDATE Z_UPDATE Z_NAVIGATE_ON_ERROR (2)

Command	Explanation	Components Activated
Z_SCRATCH	Delete Comment: (1) Only in case of error (2) Not in case of error	Z_UPDATE Z_NAVIGATE_ON_ERROR (i) Z_CLEAR_INPUT_FIELDS (2)
Z_START_KEY	Start the function with selection	Z_INITIALIZE Z_CHECK_EXISTENCE Z_LOCK_RECORD Z_FILL_DIALOG
Z_START_NKEY	Start the function without selection	Z_INITIALIZE Z_CLEAR_INPUT_FIELDS
Z_START_SOLO	Start the function without selection and not from the browse dialog	Z_INITIALIZE Z_CLEAR_INPUT_FIELDS

Associated Variables

Name	Value	Comment
LZ_CLEAR_AFTER_SAVE	**False**	Retain the input fields after saving
	True	Delete the input fields after saving
LZ_CLEAR_BEFORE_READ	**False**	Retain the input fields before reading
	True	Delete the input fields before reading

Modal Window

Description

This frame can be used to implement every kind of modal window that is called from a higher level dialog.

Access to preliminary data is not possible from this frame. It is therefore not a suitable substitute for a subdialog, but rather for optional marginal functionality which does not use much business data.

Parameters that should be passed to the frame can be supplied through the standard interface (PZ_DATA). The return of data to the calling dialog also takes place through this interface.

Links with Other Dialogs

Called:	Directly through the command system from a browse dialog
Calls:	Deletion Key dialog (required) Modal Window (handcoded link) Subdialogs (handcoded link)

Dialog Layout

Component	Comments
Push buttons	OK, Close, Help

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_CMD_EXEC_START		
Z_CMD_EXEC_END		
Z_CUSTOM_CMD		
Z_FILL_DIALOG		X
Z_INITIALIZE		
Z_NAVIGATE_ON_ERROR		
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_RETURN_PARMS		
Z_VALIDATE		

Available Commands

Command	Explanation	Components Activated
Z_CANCEL	Interrupt processing	
Z_CANCEL_KEY	Key input was interrupted	
Z_CLOSE	Close the window	Frame performance Z_VALIDATE Z_RETURN_PARMS
Z_DATA_MOD	Data has been modified	
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key	Z_RECEIVE_KEY
Z_NAV_ERR	Following batch error	Z_NAVIGATE_ON_ERROR
Z_START_KEY	Call the window with key	Z_ASSIGN_CONTROL Z_INITIALIZE Z_FILL_DIALOG
Z_START_NKEY	Call the window without key	Z_ASSIGN_CONTROL Z_INITIALIZE Z_FILL_DIALOG

Additional Information

Predefined Command Buttons

If you remove predefined command buttons from the dialog, you must also remove the corresponding instructions, which reference these dialog elements, from the Z_ASSIGN_CONTROL component.

Additional Processing

Where required additional processing, such as existence checking or storing data, can be included in the Z_VALIDATE component.

Locking Data

The frame contains a subroutine for locking data, but does not execute this processing.

If additional records should be logically locked in a subdialog, the variables LZ_LOCK_OBJ_ID and LZ_LOCK_KEY must be set and subsequently the subroutine Z_CHECK_AND_LOCK_RECORD must be called.

Data Transfer

To transfer data to the modal window or from the modal window to the calling dialog, you can use the user buffer PZ_DATA in the standard interface. If you wish to transfer data during processing to the calling dialog, then use the command SEND EVENT and include the user buffer in the data sent.

Nonstandard Dialog

Description

This frame supports navigation and command interpretation. It contains the standard interface for parameters, through which there is communication within the whole application.

Any further functionality must be implemented individually.

Links with Other Dialogs

Called	Directly through command system
Calls	None

Dialog Layout

Component	Comments
Data entry	Allows update of fields read using the single object access module, excluding the search key.
Push buttons	None

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		X
Z_INITIALIZE		
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		

Available Commands

Command	Explanation	Components Activated
Z_CANCEL	Interrupt processing	
Z_CANCEL_KEY	Key input was interrupted	
Z_CANCEL_TMR	Cancel timer activated after runtime error	
Z_CLOSE	Close dialog	
Z_DATA_MOD	Data has been modified	
Z_EXIT Z_EXIT_TMR.	End application Comments: A timer is activated that triggers the command	
Z_EXIT_TMR	Close dialog due to termination of application	
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_KEY	Key dialog sends selected key	Z_RECEIVE_KEY
Z_START_KEY	Start with key	Z_INITIALIZE
Z_START_NKEY	Start without key	Z_INITIALIZE

Associated Variables

Name	Value	Comment
LZ_FRAME_CMD_ID		Command to be processed by the frame.
PZ_SEL_KEY		Key transferred by key dialog.
PZ_KEY		Technical key.
PZ_OBJ_ID		Object type of the object to be processed.
LZ_VAL_ERR	True	During processing an error has occurred.
LZ_FOCUS		Handling of the dialog element to be activated.
LZ_DLG_TYPE		The dialog type allocated to the dialog.

Variables for Controlling Frame Behavior

The following variables for controlling frame behavior do not appear in the suggested code. Values deviating from the default are to be assigned in the customizable component Z_INITIALIZE.

- LZ_DLG_TYPE

Locking Data

The frame contains a subroutine for locking data, but does not execute this processing.

If additional records should be logically locked in a subdialog, the variables LZ_LOCK_OBJ_ID and LZ_LOCK_KEY are to be set and subsequently the subroutine Z_CHECK_AND_LOCK_RECORD is to be called.

Subdialog

Description

Subdialogs process additional attributes not included in the main dialog. Subdialogs are not modal, so for one main dialog, several subdialogs can be processed at once. The communication between the main dialog and the subdialogs is implemented internally within the frame. Modifications of the data are stored in preliminary copies of the data records.

Several modal windows can be called by a subdialog. At one time, however, only one modal window can be open. The calls to the modal windows are coded individually.

Multiple actions (add, modify, display) can be provided by one dialog.

Links with Other Dialogs

Called	From maintain dialog (manually link)
Calls	Modal window (manually link)

Dialog Layout

Component	Comments
Data entry	Allows update of fields read using the single object access module, excluding the search key.
Push buttons	OK, Cancel, Confirm, Refresh, Help

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_CMD_EXEC_END		
Z_CMD_EXEC_START		
Z_CUSTOM_CMD		
Z_FILL_DIALOG		X
Z_INITIALIZE		X
Z_NAVIGATE_ON_ERROR		
Z_READ_PREL_REC		X
Z_RECEIVE_DATA		
Z_RECEIVE_KEY		
Z_UPDATE_PREL_REC		X
Z_VALIDATE		

Available Commands

Command	Explanation	Components Activated
Z_CANCEL	Interrupt processing	
Z_CANCEL_KEY	Key input was interrupted	
Z_CLOSE	Close dialog Comments: (1) Only for update functions.	Z_VALIDATE (1) Z_UPDATE_PREL_REC (1)
Z_CONFIRM	Update preliminary files	Z_VALIDATE Z_UPDATE_PREL_REC
Z_DATA_MOD	Data has been modified	
Z_ENTER_SUB	Subdialog is activated	
Z_GET_DATA	Modal window sends data	Z_RECEIVE_DATA
Z_GET_FOCUS	Dialog is given the focus	
Z_GET_KEY	Key dialog sends selected key	Z_RECEIVE_KEY
Z_INIT	Initialization when maintain processing type changes Z_FILL_DIALOG	Z_INITIALIZE Z_READ_PREL_REC
Z_NAV_ERR	Following back error	Z_NAVIGATE_ON_ERROR
Z_OK	Update preliminary files and close dialog Comments: (1) Only for update functions	Z_VALIDATE (1) Z_UPDATE_PREL_REC (1)
Z_REFRESH	Reverse changes since last update of the preliminary copies	Z_READ_PREL_REC Z_FILL_DIALOG
Z_SAVEAS	Update dialog after ’Save as’	Z_INITIALIZE
Z_START_KEY	Start of dialog	Z_INITIALIZE Z_READ_PREL_REC Z_FILL_DIALOG

Associated Variables

Name	Value	Comment
LZ_CHECK_MODIFY	**True**	The frame controls the activation of dialog elements allocated to commands using the CHANGE- or CLICK-EVENTS of all dialog elements
LZ_GEN_TITLE	**True**	The dialog title is generated by the frame
LZ_DLG_TYPE		The dialog type allocated to the dialog
PZ_SEL_KEY		Key passed by key dialog
PZ_KEY		Technical key
PZ_OBJ_ID		Object type of the object to be processed
LZ_LOCK_OBJ_ID		Object type of the data record to be locked
LZ_LOCK_KEY		Key of the data record to be locked
LZ_VAL_ERR	True	During processing, an error has occurred
LZ_FOCUS		Handle of the dialog element to be focussed
PZ_MSG.PZ_MSG_FILL (*)		Additional information for message
PZ_MSG.PZ_MSG_NUM (*)		SYSERR error number for message
PZ_ERR_FLD_POS		Identification of the erroneous field in the interface of the access module concerned
PZ_CMD_ID		Command
PZ_CMD_TYPE_MAIN		Command type
PZ_ACT_TYPE_CUR		Current Action type
PZ_DLG_ID		Natural dialog ID
LZ_PREL_KEY		Key for access to preliminary data record
LZ_FRAME_CMD_ID		Command to be processed by the frame

Variables for Controlling Frame Behavior

The following variables for controlling frame behavior do not appear in the suggested code. Values deviating from the default are to be assigned in the customizable component Z_INITIALIZE.

- LZ_CHECK_MODIFY,
- LZ_DLG_TYPE,
- LZ_GEN_TITLE

Locking Data

The frame contains a subroutine for locking data, but does not execute this processing.

If additional records should be logically locked in a subdialog, the variables LZ_LOCK_OBJ_ID and LZ_LOCK_KEY are to be set and subsequently the subroutine Z_CHECK_AND_LOCK_RECORD is to be called.

Background Program

Description

This program is used to implement background programs which start from an online program using subprogram ZXBG010N.

It contains the standard interface for the program with which the background program communicates.

Any additional functionality must be individually implemented.

Links with Other Dialogs

Called by:	Dialogs using subprogram ZXBG010N
Calls:	None

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Implementation	Description
Z_BG_INIT		Subroutine called once to execute program dependent initialization.
Z_STORE_RESTART_DATA		Subroutine used to write restart data.
Z_BG_CODING	X	Main subroutine to implement the background functionality.

Associated Variables

Name	Description
LZ_PGM_PARM	Parameters passed from the dialog.
LZ_RESTART_DATA	Data required for a restart.

The variables are defined in the background program. They can be redefined to fit the data passed from the dialog. The variable LZ_RESTART_DATA is set during the execution of the background program. It is needed in case of a restart.

Load Objects Subprogram

Description

This subprogram is used to load objects of a specified type into the database from a workfile.

Links with Other Dialogs

Called by:	The load data dialog in the application shell.
Calls:	None

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_UL_INIT_UNLOAD		X
Z_UL_FILL_VIEW		X
Z_UL_STORE_VIEW		X#

Associated Variables

Name	Description
PZ_UL_KEY_FROM	The start key for the range of records to be loaded.
PZ_UL_KEY_THRU	The end key for the range of records to be loaded.
PZ_UL_WF_DATA(*)	The data of the record to be loaded.

Unload Objects Subprogram

Description

This subprogram is used to unload objects of a specified type from the database to a workfile.

Links with Other Dialogs

Called by:	The unload data dialog in the application shell.
Calls:	None

Customizable Components

Components marked with an "X" contain executable code while unmarked components contain only suggested code.

Name	Prototype	Production
Z_UL_INIT		
Z_UL_SELECT_TOP		X
Z_UL_SELECT_LOW		X
Z_UL_WRITE_WORKFILE		X
Z_UL_DELETE_RECORDS		X
STORE_REFERENCED_VIEW		

Associated Variables

Name	Description
PZ_UL_KEY_FROM	The start key for the range of records to be unloaded.
PZ_UL_KEY_THRU	The end key for the range of records to be unloaded.
PZ_UL_WF_DATA(*)	The data of the record to be unloaded.

Standard Commands

The application shell includes a command processor. The command processor is a background dialog which controls the processing of defined commands. You can used the attribute COMMAND-ID in the Natural dialog editor to connect some types of dialog elements with a command. The following dialog elements can be attached to commands:

- Menu item. The menu is defined in the dialog editor.
- Tool bar item. The tool bar can be defined in the dialog editor and/or assigned dynamically depending on the dialog type. The dynamic tool bar and the dialog type must be defined in the application shell.
- Push button. Push buttons can be defined in the dialog editor or added dynamically.
- Bitmap. Bitmaps can be defined in the dialog editor or added dynamically.

Standard commands are commands which are predefined in the application shell and can be processed directly by frames.

A standard command can be a local or an internal command. Although the commands of type "action", "start a function", "start a browse" and "start application" can also be processed from most frames, they are not considered standard commands (because of their type).

Not all standard commands are supported by all frames. The respective frame descriptions indicate which frames support which standard commands. The module Z_CUSTOM_CMD is called to process any standard command which is not supported by a given frame.

The following overview describes the functionality of each standard command. A complete description of the processing of standard commands can be found in the corresponding frame description.

- Local Standard Commands
- Internal Standard Commands
- Tracing a Command

Local Standard Commands

Local standard commands are defined in the application shell administration system as local commands. They are invoked by a user action, for example, selection of a menu entry.

The processing of local commands can involve the activation of multiple internal commands in other dialogs.

- Z_APPLSTART
- Z_CALL
- Z_CANCEL
- Z_CLEAR
- Z_CLOSE
- Z_CONFIRM
- Z_EXIT
- Z_HELP
- Z_HELPCNTNT
- Z_HELPUSE
- Z_INFO
- Z_INFOBUFFER
- Z_INITBUFFER
- Z_NEXT
- Z_OK
- Z_OPEN
- Z_PREVIOUS
- Z_READ
- Z_REFRESH
- Z_SAVE
- Z_SAVEAS
- Z_SCRATCH
- Z_SEARCH

Z_APPLSTART

Starts a dialog with icon-based navigation and displays the entries of the entry level dialog. This command has no effect on current processing.

Z_CALL

Opens the dialog for direct call from functions and listings. All functions and listings can be started from this dialog. This command has no effect on current processing.

Z_CANCEL

Closes the current dialog. A confirmation window is optionally possible prior to closing of the dialog, in which the end user is asked to confirm whether or not modifications are to be applied.

Z_CLEAR

Closes current processing and supplies the dialog elements of the dialog with initial values. This is needed for initialization of new areas for mass processing.

Z_CLOSE

Closes the current dialog. A confirmation window is optionally possible prior to closing the dialog, in which the end user is asked to confirm whether or not the modifications are to be applied. In this case, all validity checks applicable for the dialog as well as for the access layer will be executed.

Z_CONFIRM

Applies the modifications resulting from a maintain dialog and/or subdialogs to the preliminary copies. Prior to application, any validity checks defined for the dialog are executed. In conjunction with the command Z_REFRESH, it is possible to save the current processing status so that it can be restored if necessary. Z_CONFIRM does not result in any modifications to the original data.

Z_EXIT

Closes the entire application. It also activates a timer which is used to invoke the standard command Z_EXIT_TMR. The combination of these two commands enables a sequential closing of all open dialogs as well as any required inquiries and key ID entries.

Z_HELP

Invokes the Windows help for the current dialog. If the Help ID is not defined, control is given to the content overview. This command has no effect on current processing.

Z_HELPCNTNT

Invokes the content overview of the Windows help. This command has no effect on current processing.

Z_HELPUSE

Invokes help information on how to use Windows help. This command has no effect on current processing.

Z_INFO

Displays the information screen for the application. This command has no effect on current processing.

Z_INFOBUFFER

Opens a dialog displaying existing initialized data information. This command has no effect on actual processing.

Z_INITBUFFER

Causes a new initialization of the command data. This command has no effect on actual processing.

Z_NEXT

Closes the current processing and reads the next selected data record.

Z_OK

Accepts possible modifications and closes the dialog.

Z_OPEN

Starts a new function with the same action and object type. This command has no effect on actual processing.

Z_PREVIOUS

Closes current processing and reads the previous data record from the previously selected set.

Z_READ

Closes the current processing and reads a new data record. Depending on the frame, either the key of the current dialog will be used for reading the data record, or a key ID dialog will be offered.

Z_REFRESH

Removes the modifications resulting from the dialog (and possibly subdialogs). In addition, the dialog elements are provided anew with the values of the preliminary copies. In conjunction with the command Z_CONFIRM, it is possible to save the processing status for possible restore as necessary.

Z_SAVE

Applies the modifications to the original data. Prior to application, all validity checks defined for the dialog as well as for the access layer are executed. If a new record has been added, the key dialog is opened.

Z_SAVEAS

Opens a key dialog for input of a new key ID. The actual storage takes place as a result of an internal command sent from the key dialog to the maintain dialog.

Z_SCRATCH

Deletes the currently displayed data record.

Z_SEARCH

Accepts the user specified selection criteria and starts the corresponding search for data records. Prior to doing so, all entries in the list box are removed.

Internal Standard Commands

Internal standard commands are not defined in the application shell administration system. They are sent to other dialogs during the processing of a local command using the standard event Z_CMD_EXEC. This is required if more than one dialog is used during the processing of a command.

An internal command can also invoke multiple additional internal commands in other dialogs.

- Z_CANCEL_DLG
- Z_CANCEL_KEY
- Z_CANCEL_TMR
- Z_CONFIRM
- Z_CONF_DLG
- Z_DATA_MOD
- Z_ENTER_SUB
- Z_EXIT
- Z_EXIT_TMR
- Z_GET_DATA
- Z_GET_FOCUS
- Z_GET_GLOBAL
- Z_GET_KEY
- Z_INIT
- Z_ITEM_ADD
- Z_ITEM_DEL
- Z_ITEM_MOD
- Z_ITEM_NEXT
- Z_ITEM_PREV
- Z_KEY_MOD
- Z_LB_CLICK
- Z_LB_DOUBLE
- Z_LB_FILL
- Z_LB_SELECT
- Z_LIST_MOD
- Z_MOD_DLG
- Z_NAV_ERR
- Z_NEW_REC
- Z_REFRESH
- Z_RESET_DLG
- Z_SAVEAS
- Z_SELECT_ALL
- Z_START_NKEY
- Z_START_KEY
- Z_START_SAVE
- Z_START_SEL
- Z_START_SOLO

Z_CANCEL_DLG

Informs the maintain dialog that a subordinate subdialog has been closed.

Z_CANCEL_KEY

Informs the dialog that the key dialog was exited with the command Z_CANCEL.

Z_CANCEL_TMR

Is invoked via a timer event. The timer is activated from frames following a runtime error. Causes current processing to be interrupted.

Z_CONFIRM

Informs a subdialog that, based on a command in the maintain dialog, the processing of the local command Z_CONFIRM is to be performed.

Z_CONF_DLG

Informs the maintain dialog that the modifications of a subdialog have been confirmed with the command Z_CONFIRM.

Z_DATA_MOD

Is invoked by a change or click event for a frame-controlled dialog element. Informs the frames that the dialog data has been modified. Causes change events for frame-controlled dialog elements to be suppressed, and the process status of the frame-controlled commands to be activated or deactivated accordingly.

The frames control by default all dialog elements whose change or click event is not suppressed. The subprograms ZXXCTIGN, ZZXCTKYN and ZXXCTMON can be used to notify the frames that dialog elements are to be handled in a nonstandard manner.

If the variable LZ_CHECK_MODIFY is set to FALSE, this command will not be invoked.

Z_ENTER_SUB

Is invoked in a subdialog when the subdialog receives the focus. It ensures that the corresponding maintain dialog is in the background.

Z_EXIT

Corresponds to the local command Z_EXIT.

Z_EXIT_TMR

This internal command is invoked by an activated timer event via the command Z_EXIT. The actual processing corresponds to the local command Z_CLOSE, with the exception that at the end of processing, the command Z_EXIT is sent to the entry dialog of the next function.

Z_GET_DATA

Is sent from a modal window to the higher level dialog. It is used to return the processed data.

Z_GET_FOCUS

Informs the dialog that it should activate itself.

Z_GET_GLOBAL

Results in the updating of the global data in this dialog.

Z_GET_KEY

Is sent from a key dialog to the higher level dialog. It is used to transfer the selected key ID.

Z_INIT

Informs the subdialog that a new process is to be initialized.

Z_ITEM_ADD

Whenever this command is sent from a foreign dialog to a browse dialog, the browse dialog adds the provided data record to the list box. The data are provided to the browse dialog in the array PZ_DATA of the group PZ_LOCAL. The browse dialog determines the structure of the corresponding list box structure by calling subroutines Z_UPDATE_ITEM and Z_PASS_KEY.

If the new data record is within the value range of the currently displayed data records of the list box, it will be added. If not, it will be integrated into the list box provided that the following conditions are met:

End-of-File has been reached and the key of the data record is greater than the highest key in the list box.

The list box is empty and the switch LZ_ADD_ON_EMPTY has been set.

The variable LZ_ADD_EVERY has been set.

Z_ITEM_DEL

Whenever this command is sent from a foreign dialog to a browse dialog, the browse dialog will remove the corresponding data record from the list box. The data are provided in array PZ_DATA of group PZ_LOCAL. The browse dialog determines the structure of the corresponding list box structure by calling subroutines Z_UPDATE_ITEM and Z_PASS_KEY.

Z_ITEM_MOD

Whenever this command is sent from a foreign dialog to a browse dialog, the browse dialog will update the corresponding data record in the list box. The data are provided in array PZ_DATA of group PZ_LOCAL. The browse dialog determines the structure of the corresponding list box structure by calling subroutines Z_UPDATE_ITEM and Z_PASS_KEY.

Z_ITEM_NEXT

Whenever this command is sent from a foreign dialog to a browse dialog, the browse dialog will return the key ID of the next selected list box entry in the variable PZ_LOCAL.PZ_KEY. The calling dialog can determine via the variables PZ_LOCAL.PZ_BOD and PZ_LOCAL.PZ_EOD whether or not additional entries exist before and after the list box entry.

Z_ITEM_PREV

Whenever this command is sent from a foreign dialog to a browse dialog, the browse dialog will return the key ID of the previously selected list box entry in the variable PZ_LOCAL.PZ_KEY. The calling dialog can determine via the variables PZ_LOCAL.PZ_BOD and PZ_LOCAL.PZ_EOD whether or not additional entries exist before and after the list box entry.

Z_KEY_MOD

Is invoked by a change or click event for a frame-controlled dialog element. Informs the frames that the key ID of the dialog has been modified. Causes change events for frame-controlled dialog elements which represent key ID fields to be suppressed, and frame-controlled commands to be activated or deactivated accordingly.

Z_LB_CLICK

Is invoked from a key and browse dialog whenever the user clicks a list box item.

Z_LB_DOUBLE

Is invoked from a key dialog if the user double-clicks on a list box item. The selected list box item is then transferred to the corresponding input dialog element and sent to the higher level dialog.

Z_LB_FILL

Is invoked in a key or browse dialog if the user issues a FILL event for a list box.

Z_LB_SELECT

Is invoked in a key or browse dialog whenever a list box item is selected. For a key dialog, this command causes the data of the selected list box item to be transferred to the input dialog element.

For a browse dialog, the list box selection modification information will be sent to the mass processing dialog (if active). If no mass processing dialog is active and the switch LZ_START_USER_INPUT is not set, the actions of subtype Delete, Display, and Modify will be enabled/disabled depending on whether or not list box entries were selected.

Z_LIST_MOD

Is sent from a listing to the corresponding mass processing dialog. It informs the mass process regarding the modification of the selection set.

Z_MOD_DLG

Is sent from a subdialog to the corresponding maintain dialog, and informs the maintain dialog that the subdialog was modified.

Z_NAV_ERR

The erroneous dialog element or subdialog must be determined. Validation in a high level dialog has resulted in an error condition and the dialog element in which the error was detected is activated.

Z_NEW_REC

Causes a browse dialog to send the key ID of the newly selected data record to the corresponding mass processing dialog.

Z_REFRESH

Corresponds to the local command Z_REFRESH. It is sent from the maintain dialog to its subdialogs whenever the local command Z_REFRESH is invoked and the variable LZ_MAIN_CONFIRM_ALL is set to TRUE.

Z_RESET_DLG

Causes the modifications in a subdialog to be reset same as with the command Z_REFRESH.

Z_SAVEAS

Informs the subdialog that its processing should be switched to a new key.

Z_SELECT_ALL

In a browse dialog, the command causes all list box items to be selected.

Z_START_NKEY

Is provided during the opening of a dialog to indicate that processing is to be started without a key ID.

Z_START_KEY

Is provided during the opening of a dialog to indicate that processing is to be started with a key ID.

Z_START_SAVE

Is provided during the opening of a key dialog to indicate that a new key ID is to be provided.

Z_START_SEL

Is provided during the opening of a key dialog to indicate that an existing key ID is to be selected.

Z_START_SOLO

Is provided during the opening of a mass processing dialog, to indicate that mass processing is to be started independent of a browse dialog.

Tracing a Command

For testing purposes, you can display the current command.

To trace a command

1. Use subroutine Z_INITIALIZE.
2. Move True to LZ_GLOBAL.LZ_CMD_TRACE.
   The resulting message box displays the current command.

Customizable Components

Component	Description
Z_ACCESS_DATA	Read several records
Z_ACTIVATE_PREL_REC	Transfer preliminary data into database. Close the transaction
Z_ADD_PREL_REC	Add preliminary record
Z_ASSIGN_DEFAULT_KEY	Define preliminary key ID for a new object
Z_ASSIGN_INPUT_TO_KEY	Transfer the user input into internal variable
Z_ASSIGN_SUBDIALOG	Allocate dependent subdialogs to the higher-level dialog
Z_CHECK_EXISTENCE	Check existence of a record
Z_CLEAR_INPUT_FIELDS	Reset input fields
Z_CMD_EXEC_END	Any functionality after execution of a command
Z_CMD_EXEC_START	Any functionality before execution of a command
Z_CUSTOM_CMD	Execute any individual commands
Z_DELETE	Delete record
Z_FILL_DIALOG	Fill dialog elements
Z_FILL_ITEM	Transfer data base fields into list box
Z_INITIALIZE	Start processing
Z_LOCK_RECORD	Lock data record
Z_NAVIGATE_ON_ERROR	In case of error, set focus to field in error
Z_PASS_KEY	Pass key to internal variable
Z_PROCESS_ITEM	Process selected item
Z_READ_PREL_REC	Read preliminary record
Z_RECEIVE_DATA	Take data from modal window
Z_RECEIVE_KEY	Take foreign key from key dialog
Z_RETURN_KEY	Provide selected value to higher-level dialog
Z_RETURN_PARMS	Provide parameters (user buffer) to higher-level dialog
Z_SET_KEY_RANGE	Set key range for reading data
Z_SELECT	Transfer selected key ID into input fields
Z_UPDATE	Modify original data (if working without preliminary records)
Z_UPDATE_ITEM	Update the list box using data sent by another dialog
Z_UPDATE_PREL_KEY	Update the key ID of the preliminary copies after the ’Save as ...’ command
Z_UPDATE_PREL_REC	Update the preliminary copies
Z_VALIDATE	Validate user input

Z_ACCESS_DATA

Read multiple records.

The component usually contains the call to a multiple-object access module. After the call of the multiple-object access module, the subroutine Z_MR_ACCESS_RESULTS should be called. It sets frame variables based on the contents of the standard parameter data area of the access module. In addition, the standard error processing of a multiple-object access module is carried out by it in case of error.

For accesses which do not use the access modules created using the frame gallery, the variable LZ_REC_EOD must be set to FALSE if further data are present. The number of records found is to be assigned to the variable LZ_REC_NUM_FOUND.

Z_ACTIVATE_PREL_REC

Transfer preliminary data into the data base.

The component usually contains the call to the frame subroutine Z_ACTIVATE_PREL_STD. From this subroutine, among other things, the activation module assigned to the variable LZ_ACTIVATE_MODULE in the component Z_INITIALIZE is called.

You can, if necessary, implement your own activation logic in this component instead of the call to the frame subroutine.

Z_ADD_PREL_REC

Add preliminary record.

The frame has already preset the operation code and the key ID value for the storing of the preliminary data record. If you need values other than these, you can assign the appropriate values at the beginning of the component.

The suggested code of this component is tailored to the layout of a preliminary data record.

If you wish to add further preliminary data records, you can insert the following suggested code at the end of the component and adapt it accordingly:

```
INCLUDE ZXFXESCC      /* escape after error
RESET PZ_XAS000
      PZ_PRELIMINARY_VIEW
MOVE key       TO LZ_STD.LZ_PREL_KEY
MOVE #ZPL_SAVE TO PZ_AS_OPERATION
PERFORM new_ACCESS_PREL
```

Z_ASSIGN_DEFAULT_KEY

Definition of the preliminary key ID when adding a new record.

This component defines the preliminary key ID of a newly added object before it is stored for the first time. It is possible to assign different values to the key ID displayed in the title line and to the technical key ID.

This component is only called if the variable LZ_USE_DEFAULT_KEY is set to TRUE.

Z_ASSIGN_INPUT_TO_KEY

Transfer the user input into the variable LZ_SELECT_KEY.

This subroutine is only called if the variable LZ_START_USER_INPUT is set to TRUE.

Z_ASSIGN_SUBDIALOG

Allocate Natural objects used.

In this component, the Natural object names of the subdialogs to be managed by the frame are made known to it. In addition, a local command is allocated in a table to each subdialog.

Z_CHECK_EXISTENCE

Check existence of the record to be processed.

The frame has already preset the operation code for the existence checking. If you need another operation code, you can assign this at the beginning of the component.

The suggested code of this component is tailored to the checking of a data record.

If you wish to check further data records, you can insert the following suggested code at the end of the component and adapt it accordingly:

```
INCLUDE ZXFXESCC      /* escape after error
RESET PZ_XAS000
      PZ_MSG
MOVE LZ_XA_READ TO PZ_AS_OPERATION
MOVE key        TO P_view.xxx_ID

CALLNAT 'xxxAS00N' USING PZ_XAS000 P_view

PERFORM Z_CHECK_RSP_CHECK_EXIST
```


Z_CLEAR_INPUT_FIELDS

Reset input fields.

This component resets the values of all dialog elements and the ’linked variables’ connected to them. It is run through before the display of data newly read in and also after the command Z_CLEAR.

Z_CMD_EXEC_END

User exit for any processing after execution of a command.

This component is run through after the execution of every command. If necessary, you can code here any processing that should be run through after a command.

The variable LZ_STD.LZ_CMD_ID contains the command originally invoked, while the variable LZ_STD.LZ_FRAME_CMD_ID contains the command to be processed by the frame. Using both these variables, you can control for which command which processing should be carried out.

Z_CMD_EXEC_START

User exit for any processing before execution of a command.

This component is run through before the execution of every command. If necessary, you can code here any processing that should be run through before a command.

The variable LZ_STD.LZ_CMD_ID contains the command invoked. Using this variable, you can control for which command which processing should be carried out.

If you want to cause the frame to carry out another standard command instead of the command invoked, you can assign the required command to the variable LZ_STD.LZ_FRAME_CMD_ID in this component.

Z_CUSTOM_CMD

User exit for commands that the frame does not support.

This component is always run through when a command is invoked that is not supported by the frame of the dialog affected. The variable LZ_STD.LZ_FRAME_CMD_ID contains the command invoked. Using this variable, you can control for which command which processing should be carried out.

Z_DELETE

Delete record.

This is handled using either an access module or an activate module. If several data records are to be deleted, either several access modules or a special activate module is to be called. In case of error, error numbers are to be allocated to the array PZ_MSG_NUM and the value TRUE is to be allocated to LZ_VAL_ERR. By leaving the subroutine by ESCAPE ROUTINE, the accompanying error message is output in a message window.

Z_FILL_DIALOG

Transfer the values from the interface of the access module to the dialog elements.

In this component, the variable values in the interface of the access module are transferred into the corresponding dialog elements and the ’linked variables’ connected to them. For simple ’input fields’, this can be done by a MOVE BY NAME statement. For complicated dialog elements, such as, control boxes or option buttons, it can be necessary to code this for each field individually.

Z_FILL_ITEM

Transfer database fields into list box.

Since, in the usual case, more than one data record is read with Z_ACCESS_DATA, the record with the index LZ_REC_NUM_IND must be transferred into the frame variables LZ_BOX_ITEM and LZ_BOX_ITEM_KEY.

This is most easily handled using a MOVE BY NAME statement. For this, the two variables must be redefined to include the variables in the parameter data area of the access module.

To transfer the fields into the list box, the subroutine Z_ADD_ITEM must be called. In this way, it is possible in this subroutine to exclude data records from being transferred into the list box.

Z_INITIALIZE

Initialize processing.

This component initializes variables used. Here you can undertake any individual initialization beyond the suggested code.

Z_LOCK_RECORD

Lock records.

In this component, you lock the data records to be processed. The suggested code is so laid out that you lock the data record defined in the variable PZ_LOCAL.PZ_KEY. If you want to lock another data record, you can adapt the instructions appropriately.

If you want to lock a range of records, you must supply the variable LZ_LOCK_KEY with the beginning and the variable LZ_LOCK_KEY_END with the end of the range of keys.

If you want to lock several individual records or ranges of records, you can copy the suggested instructions and adapt them accordingly. With every call of the frame subroutine Z_CHECK_AND_LOCK_RECORD, a record or a range of records is locked.

Z_NAVIGATE_ON_ERROR

Set focus to field in error or navigate to relevant dialog after an error is detected by the access layer. Head for the erroneous dialog element or dialog after the appearance of an error in the access layer.

This component contains a DECIDE statement in which, depending on parameters passed from the access layer. The focus is set to a field in error in the current dialog or the relevant subordinate dialog receives the focus.

If you use several object views in a dialog, you should embed the DECIDE statement of the suggested code in a decide statement which checks the object type:

```
DECIDE ON FIRST VALUE OF PZ_LOCAL.PZ_ERR_OBJ_ID
  VALUE objecttype 1
    DECIDE ON FIRST VALUE OF PZ_LOCAL.PZ_ERR_FLD_POS
      VALUE ....
          :
    END-DECIDE
  VALUE objecttype 2
    DECIDE ON FIRST VALUE OF PZ_LOCAL.PZ_ERR_FLD_POS
      VALUE ....
          :
    END-DECIDE
          :
END-DECIDE
```

Z_PASS_KEY

Pass key to internal variable.

Z_PROCESS_ITEM

Carry out processing for selected list box entry. As a rule, this is achieved by calling the subroutine Z_START_FUNCTION. Alternatively, you can code any special processing.

Z_READ_PREL_REC

Read preliminary data record.

The frame has already preset the operation code and the key ID value to read the preliminary data record. If you need values other than these, you can assign the appropriate values at the beginning of the component.

The suggested code of this component reads a preliminary data record.

If you want to read further preliminary data records, you can insert the following suggested code at the end of the component and adapt it accordingly:

```
INCLUDE ZXFXESCC      /* escape after error
RESET PZ_XAS000
      PZ_PRELIMINARY_VIEW
MOVE key        TO LZ_STD.LZ_PREL_KEY
MOVE LZ_XA_READ TO PZ_AS_OPERATION
PERFORM VIEW_ACCESS_PREL
```

Z_RECEIVE_DATA

Accept data from modal window.

This component transfers data that were modified and confirmed in a modal window.

If, in a dialog, you receive different data from various modal windows, you must, when calling a modal window, note in a user-defined identifier which window was opened. When transferring the data into the component Z_RECEIVE_DATA, this identifier must be tested.

Z_RECEIVE_KEY

Accept foreign key ID from key dialog.

This component accepts foreign key IDs that were selected in a key dialog.

If, in a dialog, you call selection helps for foreign key IDs of different fields, you must, when calling a selection help, note in a user-defined identifier for which field the selection help was opened. When accepting the key ID value, depending on this identifier, the appropriate field is to be used.

Z_RETURN_KEY

Check user input for OK in the key dialog.

Validate input fields and transfer them into frame variable LZ_SELECT_KEY.

Z_RETURN_PARMS

Provide parameters (user buffer) to a higher level dialog.

In this component, the data modified in a modal window can be transferred into the user buffer PZ_DATA. Subsequently, the modal window passes the user buffer to the higher level dialog. There, the data can be received in the component Z_RECEIVE_DATA.

Z_SET_KEY_RANGE

Check start-value input and build up start value for multiple-record access module.

Example:

```
IF #IF_PLZ.STRING =  ''                  /* PLZ must be filled
   MOVE TRUE    TO LZ_VAL_ERR            /* error
   MOVE 4711    TO PZ_MSG.PZ_MSG_NUM(1)  /* message number
   MOVE #IF_PLZ TO LZ_FOCUS              /* position on input field
   ESCAPE ROUTINE                        /* no further processing
END-IF
*
*  build up start and end value for access module
*
MOVE #IF_PLZ.STRING   TO P_PERSONNEL.P_PERS_PLZ_FROM
MOVE #IF_PLZ.STRING   TO P_PERSONNEL.P_PERS_PLZ_THRU
MOVE #IF_NAME.STRING  TO P_PERSONNEL.P_PERS_NAME_FROM
```

Z_SELECT

Transfer selected key ID into input field.

This component transfers the key ID values of the selected list box entry into the appropriate input fields.

Z_UPDATE

Modify original data.

This component updates the data base according to the user input in dialogs that work without preliminary records.

Z_UPDATE_ITEM

Update the list box based on a data record sent from another dialog.

The data record is to be transferred from the array PZ_LOCAL.PZ_DATA of the standard PDA ZXXLOC0A into the appropriate region of the parameter data area of the single-record access module. Afterwards, the individual fields are transferred into the corresponding fields of LZ_BOX.

Z_UPDATE_PREL_KEY

Update the key ID of the preliminary files after the ’Save as ...’ command.

In this component, the preliminary files are read and stored with the new key ID and new processing time stamp PTS. This is always necessary when the command ’Save as ...’ is executed.

The suggested code of this component processes a preliminary data record.

If, in your function, you process several preliminary data records, you can copy the suggested code of the component for each further preliminary data record. Additionally, the following suggested code is to be inserted between the copied suggested codes and adapted accordingly.

```
INCLUDE ZXFXESCC      /* escape after error
RESET PZ_XAS000
      PZ_PRELIMINARY_VIEW

MOVE LZ_DLG.LZ_PTS_OLD TO PZ_LOCAL.PZ_PTS
MOVE LZ_DLG.LZ_KEY_OLD TO LZ_STD.LZ_PREL_KEY
MOVE #ZPL_CLOSE        TO PZ_AS_OPERATION
```

Z_UPDATE_PREL_REC

Update the preliminary files.

In this component, values of the dialog elements and the ’linked variables’ connected to them are transferred into the variables of the interface of the access layer. For simple ’input fields’, this can be done by a MOVE BY NAME statement. For complicated dialog elements, such as, control boxes or option buttons, it can be necessary to code this individually.

The frame has already preset the operation code and the key ID value for reading the preliminary data record. If you need values other than these, you can assign the appropriate values at the beginning of the component.

The suggested code of this component updates a preliminary data record.

If, in your function, you process several preliminary data records, you can copy the suggested code of the component for each further preliminary data record. Additionally, the following suggested code is to be inserted between the copied suggested codes and adapted accordingly.

```
INCLUDE ZXFXESCC      /* escape after error
RESET PZ_XAS000
      PZ_PRELIMINARY_VIEW
MOVE key        TO LZ_STD.LZ_PREL_KEY
MOVE LZ_XA_READ TO PZ_AS_OPERATION
```


Z_VALIDATE

Validate user input.

In this component, the user inputs of the current dialog are validated. If necessary, data records referenced through foreign key IDs can also be read here to display additional information to this.

Reusable Components

The frame gallery contains a number of reusable components that can expand the functionality of the generated dialog. These components are objects which use different Natural object types.

This section describes the object types that are used for each component and how each are called.

- Communication with the Command Processor
- Communication with the Data Buffer
- Starting a Dialog (Application, Function, Listing)
- Processing Status of Dialog Elements
- Message Window
- Date Validation
- Numeric Validation
  - Natural Subprogram: ZXXNC00N
- Logical Locking

Communication with the Command Processor

The control and manipulation of commands from application dialogs is performed by the command processor. Subroutines for communication with the command processor are defined in copy code ZXFXCP0C.

- Subroutine: Z_CMD_DISABLE
- Subroutine: Z_CMD_ENABLE
- Subroutine: Z_CMD_ADD_CTRL
- Subroutine: Z_SEND_CMD_PROC
- Operation: Z_CMD_CHECK
- Operation: Z_CMD_UNCHECK
- Operation: Z_CMD_DELETE
- Operation: Z_CMD_RENAME
- Operation: Z_CMD_REPLACE
- Operation: Z_CMD_DIL_REPLACE

Subroutine: Z_CMD_DISABLE

Description

Disables all dialog elements associated to a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command to be disabled

Example

```
MOVE 'Z_MODIFY' TO LZ_COMMAND
PERFORM Z_CMD_DISABLE
```

Subroutine: Z_CMD_ENABLE

Description

Enables all dialog elements associated to a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command to be enabled

Example

```
MOVE 'Z_MODIFY' TO LZ_COMMAND
PERFORM Z_CMD_ENABLE
```

Subroutine: Z_CMD_ADD_CTRL

Description

Assigns an additional dialog element (bitmap or push button) to a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_CONTROL	The handle of the dialog element
Input	LZ_COMMAND	The command to which the bitmap or push button is to be assigned. For a push button, the assignment can be omitted (command = value of the attribute COMMAND-ID)

Example

```
MOVE 'Z_MODIFY' TO LZ_COMMAND
MOVE #BM-MODIFY TO LZ_CONTROL
PERFORM Z_CMD_ADD_CTRL

MOVE #PB-MODIFY TO LZ_CONTROL
PERFORM Z_CMD_ADD_CTRL
(attribute STRING from push button is 'Z_MODIFY')
```

Subroutine: Z_SEND_CMD_PROC

Description

This subroutine is used to send all other requests to the command processor. The variable LZ_EVENT is used to indicate the processing to be performed.

Operation: Z_CMD_CHECK

Description

Places a check mark on a menu item associated to a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command

Example

```
MOVE 'Z_CMD_CHECK' TO LZ_EVENT
MOVE 'DATE' TO LZ_COMMAND
PERFORM Z_SEND_CMD_PROC
```

Operation: Z_CMD_UNCHECK

Description

Removes check mark on a menu item associated to a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command

Example

```
MOVE 'Z_CMD_UNCHECK' TO LZ_EVENT
MOVE 'DATE' TO LZ_COMMAND
PERFORM Z_SEND_CMD_PROC
```


Operation: Z_CMD_DELETE

Description

Deletes all dialog elements associated with a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command

Example

```
MOVE 'Z_CMD_DELETE' TO LZ_EVENT
MOVE 'SALARY' TO LZ_COMMAND
PERFORM Z_SEND_CMD_PROC
```

Operation: Z_CMD_RENAME

Description:

Assigns all dialog elements for a command to a new command.

The following attributes of the dialog element are also modified:

Menu Item:	STRING DIL-TEXT
Tool bar Item:	DIL-TEXT BITMAP-FILE-NAME
push button:	STRING DIL-TEXT
Bitmap:	DIL-TEXT BITMAP-FILE-NAME

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Old command
Input	LZ_COMMAND_NEW	New command

Example

```
MOVE 'Z_CMD_RENAME' TO LZ_EVENT
MOVE 'SALARY' TO LZ_COMMAND
MOVE 'VACATION' TO LZ_COMMAND_NEW
PERFORM Z_SEND_CMD_PROC
```


Operation: Z_CMD_REPLACE

Description:

Replaces place holder values for all dialog elements of a command name (attribute STRING).

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command
Input	LZ_COMMAND_FILL (1)	Value for place holder :1:
Input	LZ_COMMAND_FILL (2)	Value for place holder :2:
Input	LZ_COMMAND_FILL (3)	Value for place holder :3:

Example

```
MOVE 'Z_CMD_REPLACE' TO LZ_EVENT
MOVE 'CALCULATE' TO LZ_COMMAND
MOVE 'VACDAYS' TO LZ_FILL_TXT (1)
PERFORM Z_SEND_CMD_PROC
```

Operation: Z_CMD_DIL_REPLACE

Description

Replaces place holder values in DIL-text for all dialog elements of a command.

Parameters

Contained in the local data area ZXXSTD0L

Input/Output	Parameter Variable	Description
Input	LZ_COMMAND	Command
Input	LZ_COMMAND_FILL (1)	Value for place holder :1:
Input	LZ_COMMAND_FILL (2)	Value for place holder :2:
Input	LZ_COMMAND_FILL (3)	Value for place holder :3:

Example

```
MOVE 'Z_CMD_DIL_REPLACE' TO LZ_EVENT
MOVE 'CALCULATE' TO LZ_COMMAND
MOVE 'VACDAYS' TO LZ_FILL_TXT (1)
PERFORM Z_SEND_CMD_PROC
```


Communication with the Data Buffer

Global data from the data buffer are contained in the local data area XXGLOBL. A copy of this local data area is available as a local data area for each application dialog. Subroutines for communication with the data buffer are defined in the copy code ZXFXCD0C.

- Natural Subroutine: Z_GIVE_GLOBAL
- Natural Subroutine: Z_UPDATE_GLOBAL

Natural Subroutine: Z_GIVE_GLOBAL

Description

Requests global data from the data buffer. The current data are provided as a local copy in the local data area ZXXGLOBL.

Example

```
   PERFORM Z_GIVE_GLOBAL
```

Natural Subroutine: Z_UPDATE_GLOBAL

Description

The global data is updated in the data buffer and then is distributed to all dialogs of the application. The variable contents of the local copy of the local data area ZXXGLOBL are transferred to the data buffer.

Example

```
   MOVE 'Unknown' TO LZ_GLOBAL.LZ_UNTITLED
   PERFORM Z_UPDATE_GLOBAL
```


Starting a Dialog (Application, Function, Browse)

External Subroutine: Z_INVOKE_FUNCTION

This subroutine is used to start an application, a function or a browse function within the application.

A function can be started directly using a Function ID or via the combination of Action and Object Type.

Parameters

Group PZ_LOCAL of the parameter data area ZXXLOC0A

Only those parameters which are relevant for the call to a subroutine (reading or writing) are described. Variables not listed below should not be modified in that they contain parameter data area values which are required by each dialog.

Input/Output	Parameter Variable	Description
Input	PZ_CMD_ID	Command to be executed
Input	PZ_CMD_TYPE_MAIN	Command main type
Input	PZ_ACT_TYPE_CUR	Command sub type
Input	PZ_CMD_PARM	Command parameter
Input	PZ_SEL_KEY	Key for dialog to be started
Input	PZ_KEY_FILLED	TRUE key is filled FALSE key is not filled
Output	PZ_DLG_NAME	Natural name of the dialog
Output	PZ_DLG_ID	Natural ID of the dialog
Output	PZ_RSP	Return message value

The parameter PZ_SEL_KEY should only be completed if PZ_KEY_FILLED is set to TRUE. This is only meaningful when a function or a listing is to be started.

The following sections describe how the parameter variables are to be filled.

Application Start via Command

Parameter	Value
PZ_CMD_ID	Command ID, Command must be of the type 'Start an Application'. Command parameter indicates the Application to be started.

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
MOVE   'Z_TOP' TO PZ_LOCAL.PZ_CMD_ID
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```


Application Start without Command

Parameter	Value
PZ_CMD_TYPE_MAIN	LZ_CMD_TYPE_APPL (constant from local data area ZXX00CL)
PZ_CMD_PARM	Application ID

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
MOVE   LZ_CMD_TYPE_APPL TO PZ_LOCAL.PZ_CMD_TYPE_MAIN
MOVE   'Z_TOP' TO PZ_LOCAL.PZ_CMD_PARM
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```

Start Browse via Command

Parameter	Value
PZ_CMD_ID	Command ID, Command must be of type 'Start Browse'. Command parameter indicates the object type for which the listing is to be started.

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
MOVE   'LZ_EMPLOY' TO PZ_LOCAL.PZ_CMD_ID
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```

Start Browse without Command

Parameter	Value
PZ_CMD_TYPE_MAIN	LZ_CMD_TYPE_OBJ (constant from local data area ZXX00CL)
PZ_CMD_PARM	Object type ID for which the browse is to be started

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
MOVE   LZ_CMD_TYPE_OBJ TO PZ_LOCAL.PZ_CMD_TYPE_MAIN
MOVE   'EMPLOYEE' TO PZ_LOCAL.PZ_CMD_PARM
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```


Function Start via Command

Parameter	Value
PZ_CMD_ID	Command ID, Command must be of type ’Start a Function’. Command parameter indicates the function to be started.
PZ_SEL_KEY	Key value
PZ_KEY_FILLED	TRUE if key value is submitted

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
       PZ_LOCAL.PZ_SEL_KEY
       PZ_LOCAL.PZ_KEY_FILLED
MOVE   ’ADM-EMP’  TO PZ_LOCAL.PZ_CMD_ID
MOVE   ’4711’     TO PZ_LOCAL.PZ_SEL_KEY
MOVE   TRUE       TO PZ_LOCAL.PZ_KEY_FILLED
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```

Function Start without Command

Parameter	Value
PZ_CMD_TYPE_MAIN	LZ_CMD_TYPE_FCT (constant from local data area ZXX000CL)
PZ_ACT_TYPE_CUR	Action Type (see local data area ZXX000CL)
PZ_CMD_PARM	Function ID
PZ_SEL_KEY	Key value
PZ_KEY_FILLED	TRUE if key value is submitted

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
       PZ_LOCAL.PZ_SEL_KEY
       PZ_LOCAL.PZ_KEY_FILLED
MOVE   LZ_CMD_TYPE_FCT TO PZ_LOCAL.PZ_CMD_TYPE_MAIN
MOVE   LZ_CMD_TYPE_ACT_DEL TO PZ_LOCAL.PZ_ACT_TYPE_CUR
MOVE   ’ADM-EMP’  TO PZ_LOCAL.PZ_CMD_PARM
MOVE   ’4711’     TO PZ_LOCAL.PZ_SEL_KEY
MOVE   TRUE       TO PZ_LOCAL.PZ_KEY_FILLED
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```


Function Start via Action Change

Parameter	Value
PZ_CMD_ID	Command ID, Command must be of type action The combination of Action and current object type indicates the function to be started.
PZ_SEL_KEY	Key value
PZ_KEY_FILLED	TRUE if key value is submitted

Example

```
RESET  PZ_LOCAL.PZ_CMD_GROUP
       PZ_LOCAL.PZ_SEL_KEY
       PZ_LOCAL.PZ_KEY_FILLED
MOVE   'Z_MODIFY' TO PZ_LOCAL.PZ_CMD_ID
MOVE   '4711'     TO PZ_LOCAL.PZ_SEL_KEY
MOVE   TRUE       TO PZ_LOCAL.PZ_KEY_FILLED
PERFORM Z_INVOKE_FUNCTION PZ_LOCAL
                                #DLG$PARENT
```


Processing Status of Dialog Elements

The frames control the data modification of a function and disable dialog elements in display functions. The frame controls, by default, all dialog elements which CHANGE events are not suppressed.

The following subprograms may be used to force the frame to control dialog elements in any other way.

- Natural Subprogram: ZXXCTIGN
- Natural Subprogram: ZXXCTKYN
- Natural Subprogram: ZXXCTMON
- Natural Subroutine: Z_DIALOG_MODIFIED

Natural Subprogram: ZXXCTIGN

Parameter: HANDLE OF ANY

Description

This subprogram can be used to force the frame to ignore a dialog element, e.g. containing a start value or controlling the display mode.

Parameters

Input/Output	Parameter Variable	Description
Input	HANDLE OF ANY	Handle of dialog element

Example

```
   CALLNAT 'ZXXCTIGN' #TB-DISPLAY_ADDITIONAL_INFO
```

Natural Subprogram: ZXXCTKYN

Parameter: HANDLE OF ANY

Description

This subprogram can be used to mark a dialog element as a key component. The frame interprets any modification as a modification of the key value. (Only effective in a mass processing dialog).

Parameters

Input/Output	Parameter Variable	Description
Input	HANDLE OF ANY	Handle of dialog element

Example

```
   CALLNAT 'ZXXCTKYN' #IF-CUSTOMER_ID
```


Natural Subprogram: ZXXCTMON

Parameter: HANDLE OF ANY

Description

This subprogram can be used to mark a dialog element as a dialog element. The frame interprets any modification as a modification of the data to be processed.

Parameters

Input/Output	Parameter Variable	Description
Input	HANDLE OF ANY	Handle of dialog element

Example

```
CALLNAT 'ZXXCTMON' #SB-CUSTOMER_TYPE
```

Natural Subroutine: Z_DIALOG_MODIFIED

Description

Forces dialog status to "modified" in the following frames: maintain, subdialog, modal windows, and nonstandard dialog.

Parameters

None.

Example

```
PERFORM Z_DIALOG_MODIFIED
```

Message Window

With Windows applications, messages are usually provided in a message window or a status line. The copy code ZXFXMSGC defines a subroutine which is used to control messages. Message texts are stored in a Natural message file. Multiple language support as well as modification of messages independent of program logic is thereby possible.

A message text is constructed as follows:

```
text:1:text:2:text:3:text
```

The place holders :1., :2: and :3: are set dynamically during runtime.

A place holder can occur multiple times in a single message.

A new line within a message is created by entering two colons in succession (::).

Natural Subroutine: Z_DISPLAY_MESSAGE

Description

This subroutine is used to display messages in a message window or in a status line of the current window. Up to 3 message texts (one line per text) can be displayed at the same time in a message window.

Parameters

Group PZ_MSG of the parameter data area ZXXMSG0A

Input/Output	Parameter Variable	Description
Input	PZ_MSG_TYPE	Message type (see below)
Input	PZ_MSG_NUM (3 occurrences)	Message numbers
Input	PZ_MSG_FILL (3 occurrences)	Place holders
Input	PZ_MSG_TITLE	Optional title for message window
Output	PZ_MSG_BUTTON	Active buttons for message window
Input	PZ_MSG_DLG (not part of the group)	Handle of parent window Usually #DLG$WINDOW)

The following message types are defined as constants in the local data area ZXX000CL:

Constant	Description
LZ_MSG_BUTTON_OK	OK button
LZ_MSG_BUTTON_YES	YES button
LZ_MSG_BUTTON_NO	NO button
LZ_MSG_BUTTON_CANCEL	CANCEL button

Example

```
Message Text 1500:       :1: employees live in :2:
```

```
MOVE 'BERLIN' TO #CITY

CITY_FIND.
FIND NUMBER EMPLOYEES WITH CITY = #CITY

MOVE LZ_MSG_TYPE_INFO TO PZ_MSG_TYPE
MOVE 1500 TO PZ_MSG_NUM(1)
MOVE *NUMBER(CITY-FIND.) TO PZ_MSG_FILL(1)
MOVE #CITY TO PZ_MSG_FILL(2)
PERFORM Z_DISPLAY_MESSAGE
```

Date Validation

Natural Subprogram: ZXXDATEN

Parameter Data Area: ZXXDATEA

Description

The subprogram verifies that the input date is a valid date. The input date can be in any of the Natural formats A10, A8, N8 or D. These formats are all available for the output date following a successful validation.

An alphanumeric date (A10 and A8) is checked according to the current date format.

The following rules apply for input dates of formats A8 and A10:

- the year can be omitted
- the century can be omitted when specifying the year
- leading zeros can be omitted for day and month entries

Parameters

Group PZ_DATE of the parameter data area ZXXDATEA

Input/Output	Parameter Variable	Description
Input	PZ_DATE_FORMAT	Date format for alphanumeric input and output date (see below) (default LZ_FORMAT_DTFORM)
Input	PZ_DATE_CONVERT	Variable format of input date (see below)
Input	PZ_DATE_IGNORE_DELIM	Only for alphanumeric input date FALSE: only the seperation character of date format is permitted (default) TRUE: any character is permitted as separation character
Input	PZ_DATE_CENTURY_LIMIT	Only for alphanumeric input date Century limit for 2 position century input (default = 0) Year. < Century Limit = 20th C. Year. >= Century Limit = Current C.
Input/Output	PZ_DATE_A10	Date using format A10
Input/Output	PZ_DATE_A8	Date using format A8
Input/Output	PZ_DATE_N	Date using format N8 (YYYYMMDD)
Input/Output	PZ_DATE_D	Date using format D
Input/Output	PZ_DATE_RSP	Response Code 0: Date OK 1: Date invalid 2: PZ_DATE_FORMAT invalid 3: PZ_DATE_CONVERT invalid

The valid values for PZ_DATE_FORMAT are defined as constants in the local data area ZXXDATCL.

Constant	Representation A8	Representation A10
LZ_FORMAT_US	MM/DD/YY	MM/DD/YYYY
LZ_FORMAT_GERMAN	DD.MM.YY	DD.MM.YYYY
LZ_FORMAT_EUROPE	DD/MM/YY	DD/MM/YYYY
LZ_FORMAT_INTERNATIONAL	YY-MM-DD	YYYY-MM-DD
LZ_FORMAT_DTFORM	according to Natural parameter DTFORM	according to Natural parameter DTFORM

The valid values for PZ_DATE_CONVERT are defined as constants in the local data area ZXXDATCL.

Constant	Variable Format for Input Date
LZ_CONVERT_A8	A8
LZ_CONVERT_A10	A10
LZ_CONVERT_N	N8
LZ_CONVERT_D	D

Example

```
MOVE LZ_FORMAT_GERMAN   TO PZ_DATE_FORMAT
MOVE LZ_CONVERT_A10     TO PZ_DATE_CONVERT
MOVE '1.3.95'           TO PZ_DATE_A10

CALLNAT 'ZXXDATEN'  PZ_DATE
```

Output

```
PZ_DATE_A10   01.03.1995
PZ_DATE_A8    01.03.95
PZ_DATE_N     19950301
PZ_DATE_D     Natural internal date format
```


Numeric Validation

Function

To validate and convert numerical values.

Name:	ZXXNC00N
PDA:	ZXXNC00A

Natural Subprogram: ZXXNC00N

Parameter Data Area: ZXXNC00A

Description

The subprogram validates and converts numerical values.

Parameters

Group PZ_NC_PARMS of the parameter data area ZXXNC00A

Input/Output	Parameter Variable	Description
Input	PZ_NC_OP	Operation code
Input/Output	PZ_NC_VALUE	Value to be processed
Input/Output	PZ_NC_BEFORE_DC	Length before the decimal character (maximum 27)
Input/Output	PZ_NC_AFTER_DC	Length after the decimal character (maximum 27)
Input	PZ_NC_DC	Decimal character
Input	PZ_NC_RSP	Response codes

The following operations are defined in local data area ZXXNC0CL:

Operation	Description
LZ_NC_CHECK	Checks the numerical input value. The value can contain a maximum of one decimal character.
LZ_NC_GIVE_LEN	Supplies the length of the input value (both before and after the decimal character).
LZ_NC_EXACT_LEN	Checks the input value against the corresponding defined length.
LZ_NC_LEN_NO_ZERO	Checks the input value against the corresponding defined length, whereby no leading zeros are allowed.
LZ_NC_FILL_LEADING	Fills the input value according to the definition with leading zeros.
LZ_NC_FILL_FOLLOWING	Fills the input value according to the definition with trailing zeros (after the decimal character).
LZ_NC_FILL	Fills the input value according to the definition with leading zeros and trailing zeros (after the decimal character).

Example

```
RESET PZ_NC
MOVE LZ_NC_CHECK   TO PZ_NC_OP
MOVE '27'          TO PZ_NC_VALUE
MOVE '3'           TO PZ_NC_LEN_BEFORE_DC

CALLNAT 'ZXXNC00N'  PZ_NC_PARMS
```


Logical Locking

Natural Subroutine: Z_CHECK_AND_LOCK_RECORD

Description

Determines if a key or key area is locked by another transaction, and, if possible, locks the key or key area.

Parameters

The parameters are contained in the local data area ZXXSTD0L.

Input/Output	Parameter Variable	Description
Input	LZ_LOCK_OBJ_ID	Identifier of the object type
Input	LZ_LOCK_KEY	The key to be locked or the beginning of the key area to be locked
Input	LZ_LOCK_KEY_END	The end of the key area to be locked
Output	LZ_VAL_ERR	TRUE: Locking is not possible

If LZ_VAL_ERR has a setting of TRUE, the subroutine also inserts the corresponding error number into the variable PZ_MSG.PZ_MSG_NUM(1).

Example

```
MOVE  'ARTICLE'        TO LZ_LOCK_OBJ_ID
MOVE  PZ_LOCAL.PZ_KEY  TO LZ_LOCK_KEY

PERFORM Z_CHECK_AND_LOCK_RECORD

IF LZ_VAL_ERR
 BACKOUT TRANSACTION
 PERFORM Z_DISPLAY_MESSAGE
ELSE
 END TRANSACTION
END-IF
```


Background Processes

Background processing can be implemented in situations where processing requirements could cause significant response time problems during dialog operation.

This is usually the case if a large number of database accesses or other time consuming processing is necessary.

Typical examples of this type of processing are complex deletion procedures, complex calculations or modifications, which require access to a large number of data records.

Depending on specific requirements, background processes can either be started from an online dialog or from the computer center.

The background program frame handles parameter transfer from the dialog system, error correction as well as components to restart background processes in the event of an abnormal termination.

The following functions are provided for the end user:

- online monitoring of background program execution;
- online display of errors detected during background processing;
- restart background processing following abnormal termination.

Background processes which are not implemented with the appropriate background frames cannot use the features described above.

The following topics are covered below:

- General Information
- Creating and Maintaining Background Procedures
- Invoking Background Processes
- Start Background Program from Dialog
- Implementing Background Programs
- Implementing Computer Center Background Processes

General Information

Background procedures must be defined in the application shell for the background process to which the background program belongs.

To invoke the background process, the subprogram ZXBG010N is called by the online program. For further information see the corresponding procedure description.

This subprogram ensures that the parameters are transferred using a control record in a control file. This control record is identified by a unique key (the timestamp).

The timestamp is passed as a standard parameter to the batch program.

The background process is then submitted to the operating system via a standard interface. The background procedure is therewith submitted to the operating system from the dialog program. The background program is then started.

The writing of a control record in the file Z_BG_PARM as well as the starting of the background process are executed from the subprogram ZXBG010N.

Therefore, it is necessary to ensure that the parameters are correctly set for the call and that the call to the subprogram ZXBG010N is coded.

Creating and Maintaining Background Procedures

General

For each background process, operating-system-dependent background procedures are required.

Note:
Background procedures must be created prior to implementing background programs. The parameters for the background procedures are required during the dialog call to the background process.

Using Administration Functions

The application shell is used to define background procedures.

Use the functions Add Background Procedure, Modify Background Procedure, etc., to create and maintain the background procedure.

For a full description, see the Natural Application Shell documentation.

Types of Background Processing

A background procedure is not needed for each background process. Instead, the various types of background processing should be identified and a background procedure for each corresponding type is created.

The following are examples of various types of background processes:

- receipt of data from another system;
- transfer of data to another system;
- copying/deleting/modifying mass volumes of data;
- printing of lists;
- loading/unloading data.

Invoking Background Processes

Calling a Background Program from a Dialog

A background program can be started from any dialog.

To call the background program from a dialog

- Enter CALLNAT ’ZXBG010N’ USING PZ_BG_START_BATCH.
  This CALLNAT contains END TRANSACTION and BACKOUT TRANSACTION statements.

Note:
When writing programs which update data, you must ensure that the procedure is called at a point at which it will not adversely affect the transaction logic of the dialog.

Parameter Usage

Both the background program and the background procedure needed for the execution of the background program require certain parameters. These parameters must be passed by the dialog whenever the subprogram ZXBG010N is invoked.

The required parameters can be displayed with the application shell function Display Background Procedure.

The parameters for the background program, for example, selection/sort criteria, which are assigned values in the online program, must also be passed.

Detailed information on these parameters is contained in the procedure description.

Start Background Program from Dialog

The subprogram ZXBG010N ensures that at runtime the parameters are transferred to the control file, from which they can be read by the background program.

Natural Object Name:	ZXBG010N
Parameter:	ZXBG010A (PDA)

Parameters

Group PZ_BG_START_BATCH of the PDA ZXBG010A.

Input/Output	Parameter Variable	Description
Input	PZ_BG_PTS	
Output	PZ_BG_RSP	0 - Ok 1 - Background procedure not found 2 - Background parameter not found (restart) 3 - Background parameter not stored 9 - Error during submit call
Output	PZ_BG_MSG_NUM	
Output	PZ_BG_MSG_FILL	
Input	PZ_BG_DELIM	Input delimiter
Input	PZ_BG_CLIENT_ID	Client ID
Input	PZ_BG_US_ID	User ID
Input	PZ_BG_PSW	Password
Input	PZ_BG_NAME	Background process name or title (if applicable for the operating system)
Input	PZ_BG_LIB	Library in which program will run
Input	PZ_BG_PGM	Program name to be executed
Input	PZ_BG_NATPARM	Natural parameter module
Input	PZ_BG_PRIORITY	Priority of the background procedure (if applicable for the operating system)
Input	PZ_BG_ONLINE_LIB	Library from which the start of the background process is invoked
Input	PZ_BG_ONLINE_PGM	Program to be called to start the background process
Input	PZ_BG_FU_ID	Function ID
Input	PZ_BG_DESCR_LC	Description
Input	PZ_BG_LA_ID	Language ID
Input	PZ_BG_LA_NAT_CODE	Natural language code
Input	PZ_BG_CD_ID	ID of the background procedure
Input	PZ_BG_PRINTER (1:5)	Printer name
Input	PZ_BG_WORKFILE (1:5)	Work file path and name
Input	PZ_BG_SUST_NAME?(1:5)	Name of substitution variable
Input	PZ_BG_SUBST_VALUES (1:5)	Content of substitution variable
Input	PZ_BG_RESTART	Restart indicator
Input	PZ_BG_PGM_PARM (1:5)	Parameter to be passed to the background program

Implementing Background Programs

The frame gallery provides three application frames for background processing:

- Background program.
- Load objects.
- Unload objects

These are described in section Application Frames.

This section provides additional information on the use of the background program frame.

- Passing Parameters
- Logically Locking Data Records
- Restart
- Error Handling
- Setting the Processing Status
- Monitoring Program Execution

Passing Parameters

The background program receives a timestamp as an input parameter.

This timestamp is used to read the corresponding control record from the control file Z_BG_PARM. This is a part of the frame functionality.

The parameters from the online program, for example, selection criteria, are made available. The transfer of parameters is thereby automatic.

The parameters are available to the background program via the variable LZ_PGM_PARM.

Logically Locking Data Records

Data records can be logically locked by background programs.

This may be necessary, for example, if the data must be modified by the background program, and at the same time processed by the dialog system.

Locking can also be required for read access. For example, a statistic which is based on certain data values requires that these data records remain unchanged during execution of the calculation of the statistic.

Locking Data Records

Use the inline subroutine Z_CHECK_AND_LOCK_RECORD to lock individual data records, data areas or objects, as required within a business application.

After execution of this subroutine, an END TRANSACTION must follow. The program must include the necessary processing logic.

The parameters are contained in the LDA ZXFBA00L.

Input/Output	Parameter Variable	Description
Input	LZ_LOCK_OBJ_ID	Identifier of the object type
Input	LZ_LOCK_KEY	The key of the record to be locked, or the beginning of the record range to be locked.
Input	LZ_LOCK_KEY_END	The end of the record area to be locked.
Output	LZ_VAL_ERR	TRUE: Locking is not possible.

If LZ_VAL_ERR has a setting of TRUE, the subroutine also inserts an error number into the variable LZ_MSG_NUM(1).

Example

```
MOVE  'ARTICLE'        TO LZ_LOCK_OBJ_ID
MOVE  PZ_LOCAL.PZ_KEY  TO LZ_LOCK_KEY

PERFORM Z_CHECK_AND_LOCK_RECORD

IF LZ_VAL_ERR
  BACKOUT TRANSACTION
  PERFORM Z_TERMINATE_PROCESS
ELSE
  END TRANSACTION
END-IF
```

Releasing Data Records

The background program frame does not contain any locking logic. Therefore, if you use the procedure for locking data records, you must also release the records following completion of processing.

Use the inline subroutine Z_CANCEL_LOCK_RECORD for this purpose.

After execution of this subroutine, an END TRANSACTION must follow. The program must include the necessary processing logic.

Restart

A background program can be restarted in the event it is terminated abnormally, provided that the portion of the processing which has been successfully completed must not be repeated.

Restart logic is recommended when:

- processing involving data modifications is to be executed, or
- extensive list processing is to be executed.

As a prerequisite for implementing restart logic, the restart points must be logged at runtime.

This entry is written immediately prior to execution of an END TRANSACTION statement and is confirmed together with the data modifications via the END TRANSACTION statement.

Use the inline subroutine Z_STORE_RESTART_DATA to write the restart data.

The call is contained in the background program frame. The data are written following logical transactions to the file Z_BG_PARM.

The number of transactions following which an END TRANSACTION statement is to be executed can be set using the variable C#TRANSACTION. The default value for this variable is 99.

Before calling this procedure, the desired restart point, for example, the key value of a database record, must be provided in LZ_RESTART_DATA(*).

The restart of the abnormally terminated background program is then performed via the application shell. For further information, see the Natural Application Shell documentation.

With this function you can view/modify existing restart data, and then restart the background process.

The frames of the batch programs ensure that the restart data provided in the variable LZ_RESTART_DATA(*) are received and are available for restart processing.

Error Handling

Errors which occur during background program execution can be logged. Differentiation between the following types of errors must be made:

- Natural runtime errors
- Application errors

Natural Runtime Errors

Natural runtime errors can be detected in the background program with the ON ERROR statement and automatically displayed in the application shell ’’Maintain Error Log" dialog.

All database modifications which were executed during the current, not yet successfully closed logical transaction, are backed out of the database and the corresponding logical locks are released. This is a part of the frame functionality.

Application Errors

Application errors can be handled in various ways:

- termination of the background program with an entry in the error log file. The inline subroutine Z_TERMINATE_PROCESS can be used for this purpose. Error handling is the same as that for Natural runtime errors.
- continuation of the background program with an entry (warning) in the error log file. The inline subroutine Z_STORE_ERROR_LOG can be used for this purpose. After execution, an END TRANSACTION must follow. In this case, the program can be closed with the status ’Process has ended with an error/warning’. This is performed by the frames when the variable LZ_APPL_ERR is set to TRUE.

In each case, these subroutines must be supplied with the variables ZER_USER_DESC(1:4) and ZER_APPL_ERR_NUM.

Display Error Log

Error logs can be displayed using the application shell function Browse Error Log.

For further information, see the Natural Application Shell documentation.

Setting the Processing Status

A background program is assigned a status when it is started from a dialog. This status can be displayed via the application shell function Browse Background Process.

Terminating a Background Program

Normally the appropriate end status is set whenever the background program is ended. This status can however be directly set to: ’Process ended with an error/warning’ by setting the variable LZ_APPL_ERR to TRUE.

Termination by Calling another Background Program

If a background program was called from another background program, and control is to be returned to the calling program, the background status may not be modified.

This can be done by setting the variable LZ_CONTINUE to TRUE.

The last program of the background process will set the status to ENDED.

Monitoring Program Execution

The application shell provides various functions for monitoring background processes. The following information is provided:

Function	Information
Browse Background Process	Status of user’s background processes
Browse Error Log	Display errors

No access is available to the operating system itself, i.e., direct intervention with executing background processes is not possible.

Implementing Computer Center Background Processes

No frame is available to implement background programs which are to be started by computer center personnel.

Note:
The background program is not intended for this purpose. It is intended only for the implementation of batch programs which are to be started from a dialog program.

Error Handling

Natural runtime errors should be handled using the ON ERROR statement.

Monitoring Program Execution

The monitoring of batch jobs (status, restart) is not supported by the application shell.

The Command System

By modifying the data contained in the application shell system files, you can define the behavior and appearance of systems you develop with the frame gallery. Because this information is stored centrally in files, it can be easily updated. Handling for multiple-language applications and access protection can be easily maintained using the administration system provided in the application shell instead of being programmed explicitly within all dialogs.

The following topics are covered below:

- Information Objects and Application Components
- Data Buffer
- Access Protection
- Command Processor
- Command Processing Description

Information Objects and Application Components

The following table lists the application components influenced by the specific information objects:

Information Object	Application Component	Used to Define
Command	Menu item	Name DIL-text
	Tool bar item	Bitmap DIL-text
	Bitmap	DIL-text
	Push button	Possible Name DIL-text
	Direct Call	Name (action)
Object Type	Icon-based Navigation	Name DIL-text
	Direct Call	Name
Function	Icon-based Navigation	Name DIL-text
	Main dialog for function	String
Application	Icon-based Navigation	Name DIL-text
	MDI-frame	String

Data Buffer

The most frequently used data are read into main storage at the start of an application, thereby reducing the number of database accesses to the application shell system files as well as reducing the overall network overhead.

This storage area (data buffer) is implemented as a transparent background dialog.

In order to access data in the data buffer, dialogs send requests (events) to this background dialog, which responds via a parameter interface.

The following data are contained in the data buffer:

Information Object	**Data Fields**
Commands (up to 400)	Command ID
	Command Type (main and subtype)
	Name
	DIL-text
	Bitmap
	Parameter
Tool bars (up to 70)	Tool bar ID (up to 30 tool bar items (commands))
Dialog Types (up to 70)	Dialog Type ID
	Tool bar ID
Object Types (up to 250)	Object Type ID
Functions (up to 800)	Function ID
	Up to 3 actions (command from main type action)
	Object ID
Applications (up to 80)	Application ID

Access Protection

Access protection is provided at the function level. A function is either permitted or prohibited for a given user.

The access to object types and applications is controlled by the access protection at the function level.

An object type (and its listing) is only permitted if at least one permitted function exists for this object.

An application is only permitted if at least one of its children (application, function, object type) is permitted.

This means that a user can only see what is authorized via the functional access protection. This has the following implications within an application:

- Menu items, tool bar items, bitmaps and push buttons, which are associated with a command, will be omitted if they are not permitted for a user. Only commands from the main type Action, Application Start, List Start and Function Start are affected by access protection.

Command Type	Prohibited
Action	Function is not permitted for current object type and action
Function Start	Function is not permitted
List Start	Object type is not permitted
Application Start	Application is not permitted

- Submenus are omitted for which no permitted command exists.
- The graphical navigation only offers the applications, functions and object types for selection which are permitted.
- The "Direct Call" dialog offers only object types for which at least one function is permitted.
- The "Direct Call" dialog displays only actions (commands from main type action) for which a function for the selected object type is permitted.

Access protection can be enabled or disabled for an entire application or for individual users.

Access Protection for Application	Access Protection for User	Check Authorization for Functions
Yes	Yes	Yes
Yes	No	No
No	Yes	No
No	No	No

Undefined users can only start an application if access protection for the application is disabled. Only those functions are checked for which access protection is enabled. Functions without access protection are available to all users.

If authorizations for functions are to be determined, the ordering of functions to function groups and the assignment of function groups to users should be considered. A function group can have permitted and/or prohibited functions.

This and the access protection defaults for the application thereby provide the set of permitted functions for a user.

Default Value of the Application	Set of Permitted Functions
All functions permitted	All functions of the application, except those functions which are prohibited via a function group. Plus those functions which are permitted via a function group.
All functions prohibited	All functions which are permitted via a function group, except functions which are prohibited via a function group, plus functions without access protection.

During start-up of the application, all permitted functions are determined based on information in the data buffer.

This in turn determines the permitted object types (see above), which are placed into the data buffer.

Using this information, the application structure is checked and the permitted applications (see above) are placed in the data buffer.

All commands of the application are read into the data buffer, whereby prohibited commands (Function Start, Application Start, Object Start) are marked.

Whether or not an action is permitted can only be determined at runtime. The action will be permitted only if the action and the current object combination referenced in the current dialog is contained in the data buffer.

Command Processor

The command processor is a background dialog which controls the processing of defined commands. Command are activated via menu items, tool-bar buttons, push buttons, and icons with which a command ID has been associated.

The dialog communicates with the command processor via events.

During the initialization of a dialog, the following tasks are executed:

- If the value of the variable LZ_SECURITY is True, it is checked whether the commands attached to the dialog elements are permitted. If a command is prohibited, the corresponding dialog elements are deleted.
- Depending on the value of the variable LZ_STRING_REPLACE, the STRING- and DIL-attributes are assigned the corresponding values of the command entries stored in the data buffer.

The tasks below are dialog element specific:

Menu Items

Empty submenus and superfluous separators are removed.

Tool Bar Items

The BIT-MAP-FILE-NAME for the dynamically added tool bar items are read from the data buffer. Superfluous separators are removed.

If the tool bar items are defined in the dialog editor and a dialog type is assigned to the dialog, the corresponding tool bar items are added dynamically behind the existing items.

Bitmaps

The BIT-MAP-FILE-NAME is read from the data buffer.

For the application programmer, the command processor is the most significant interface for controlling dialog elements with command ordering.

If, for example, a command must be disabled, it is only necessary to indicate this to the command processor. The associated dialog elements are then determined and disabled by the command processor.

The following functions for commands and the associated dialog elements are available:

Function	Affected Dialog Element
Enable	Menu item, tool bar item, push button, bitmap
Disable	Menu item, tool bar item, push button, bitmap
Set marker	Menu item
Remove marker	Menu item
Rename	Menu item, push button
Replace (Attribute String)	Menu item, push button
Replace (:n:) position holder	
Replace (attribute DIL-TEXT)	Menu item, tool bar item, push button,
Replace (:n:) position holder	bitmap
Delete	Menu item, tool bar item, push button, bitmap

Command Processing Description

This section describes the processing of dialog elements which are assigned to a command.

Command processing consists of the following steps:

1. Assignment of commands to dialog elements via the attribute COMMAND-ID.
2. Clicking of a dialog element which is assigned to a command.
3. Retrieval of associated command data via the command processor.

Command	Variable
ID	LZ_FRAME_CMD_ID
Main type	LZ_FRAME_CMD_TYPE_MAIN
Sub type	LZ_FRAME_CMD_TYPE_SUB
Parameter	LZ_FRAME_CMD_PARM

4. Call to user subroutine Z_CMD_EXEC_START.
   Additional processing prior to standard processing is possible here, for example, to change the command.
5. Standard processing Z_CMD_EXEC_FRAME.
   Standard processing of the command. If the command cannot be processed by frames, the user subroutine Z_CUSTOM_CMD is called.
6. Call to user subroutine Z_CMD_EXEC_END.
   Additional processing subsequent to standard processing can be performed at this point.

List Box Handling

This section describes how to maintain two list boxes and the communication between them. The first list box displays a set of objects from which entries can be selected and transferred to the second list box.

- **List box ALL**
  Contains all available objects.
  Example: All articles required for an order.
- **List box SELECTED**
  Contains all objects selected from list box ALL.
  Example: All articles selected to create an invoice.

The following topics are covered below:

- Prerequisites
- Functional Scope of the Frame Modules
- Additional User Activities
- Integrate Processing into the Dialog

Prerequisites

The following modules for the frame modules are prerequisites for list box processing:

- **ZXFXLB0C** - This copycode contains the general code for the integration of list box processing in a dialog.
- **ZXFXLB1C** - This copycode contains the special-purpose code for the integration of list box processing in a dialog.
- **ZXFXLB2C** - This copycode contains the general code for the integration of list box processing in a subprogram.
- **ZXFXLB0N** - This subprogram is the skeleton for the creation of list box processing.
- **ZXFXLB0L** - This local data area contains all variables required by the subprograms for list box processing.
- **ZXFXLB0A** - Standard parameter data area for list box processing for the subprogram.
- **ZXFXLB01A** - Standard parameter data area for list box processing for list box ALL. In addition, this parameter data area is the basis for the creation of individual interfaces.
- **ZXFXLB02A** - Standard parameter data area for list box processing for list box SELECTED. In addition, this parameter data area is the basis for the creation of individual interfaces.

In addition, the access modules and their parameter areas are required to fill and process both list boxes. Multiple-record access module and the parameter for the object type whose occurrences are displayed in the list box. For example: Item.

Single-record access module and the parameter for the object type for which selections are carried out. For example: Invoice.

Functional Scope of the Frame Modules

The following functionality is provided by the frame modules:

- Search using a Start Value in List Box ALL.
  For the list box ALL, which may have a large number of entries, a start value can be provided. When the Search button is selected, the list box is displayed beginning with the specified start value.

- Dynamic read for list box ALL.
  To minimize the response time for large data volumes, list box ALL is at first not filled with all possible entries. Instead, the data is dynamically read as required when the user pages downwards within the list.

- Analysis of the Selection in list Box ALL.
  The following is checked whenever a user selects one or more entries in the list box ALL:
  - Maximal number selected?
    Does list box SELECTED already contain the maximum permitted entries?
    If it does, no further selection is possible and the Select button is not activated.
    Example: The frame guarantees that no more than 50 items can be selected for one invoice.
  - Double selection?
    Does list box SELECTED already contain the entries selected in list box ALL?
    If it does, repeated selection is not possible and the Select button is not activated.
    Example: The frame guarantees that each item can be selected only once per invoice.
  - Selection
    All selected entries in list box ALL are transferred to list box SELECTED - if not previously done - by using the Select button or a double-click on the list box entry.

The following options are available when adding new entries to the list box SELECTED:

- Insert in sort sequence.
  New entries are inserted in the sorting sequence if the list box was implemented as a sorted list box (Mark attribute SORTED in the dialog editor).

- Insert at end of list.
  New entries are added at the end of list, when the list box #LB_SELECTED is not implemented as a sorted list box (attribute SORTED in the dialog editor is not marked) and the user has not selected an entry in this list box.

- Insert at a specific position within a list.
  New entries are inserted at a specific position, when the list box SELECTED is not implemented as a sorted list box (attribute SORTED in the dialog editor is not marked) and the user has selected an entry in this list box. In this case, the new entry is inserted before the selected entry.
- Remove.
  The selected entries in list box SELECTED are removed when the Remove button is selected.

Additional User Activities

Subprogram

Copy the subprogram ZXFXLB0N to subprogram xxxXLB0N and adapt it as described in the suggested code.

Copy the parameter data area ZXFXLB1A to parameter data area xxxXLB1A.

Copy the parameter data area ZXFXLB2A to parameter data area xxxXLB2A.

Copycode

Copy copycode ZXFXLB1C to copycode xxxXLB1C and adapt it as described in the suggested code. Thus the inline subroutine PROCESS_LISTBOX_xx_yy is created.

Dialog Layout

Create the following graphical elements for a dialog:

List Box ALL

Name HANDLE according to copycode xxxXLB1C.

Mark the attribute MULTIPLE SELECTION if several entries are to be selected at one time.

Call subroutine PROCESS_LISTBOX_xx_yy in the event handlers for CLICK, DOUBLE-CLICK and FILL.

List box SELECTED

Name HANDLE according to copycode xxxXLB1C.

Mark the attribute MULTIPLE SELECTION if several entries are to be selected at one time.

Call subroutine PROCESS_LISTBOX_xx_yy in the event handlers for CLICK and DOUBLE-CLICK.

Search Push button

Assign COMMAND-ID according to copycode xxxXLB1C.

Add Push button

Assign COMMAND-ID according to copycode xxxXLB1C.

Delete Push button

Assign COMMAND-ID according to copycode xxxXLB1C.

Assign Data Areas

In addition, assign the following global data areas and parameter data areas:

- ZXFXLB0A
- xxxXLB1A
- xxxXLB2A

Include Copycode

Place copycode xxxXLB1C in the AFTER ANY event of the dialog.

Integrate Processing into the Dialog

Subroutine Z_INITIALIZE

Add the following line:
INCLUDE ZXFXLB4C

Subroutine Z_FILL_DIALOG

Add the following lines:
INCLUDE ZXFXLB3C
PERFORM PROCESS_LISTBOX_xx_yy

Subroutine Z_CUSTOM_CMD

Call subroutine PROCESS_LISTBOX_xx_yy for commands *Search*, *Add*, and *Delete*

Subroutine Z_UPDATE_PREC_REC

To re-transfer data from list box SELECTED to the respective access module parameter data area, add the following lines at the respective position:

MOVE LZ_LB_RETURN TO PZ_LISTBOX.PZ_OPERATION

PERFORM PROCESS_LISTBOX_xx_yy

Creating Object Views

With Client/Server technology, it is increasingly important that data access is encapsulated in an object-oriented manner. In addition to enabling simple data access to be switched from one computer to another, this modularization offers the following benefits:

- Each access must be coded only once.
- An access operation, when available and tested, can be eliminated as a source of error.
- All object operations are located in a clearly defined place
- Database changes result in minimum modification in easily locatable places.
- An access operation can be used wherever it is required within a system.
- When naming conventions are adhered to, required access operations are easily found.

The following topics are covered below:

Concepts

All database accesses are encapsulated in object views. The set of object views for an application builds the access layer for the application. The associated "object orientation" makes it possible to distribute data using a Client/Server approach.

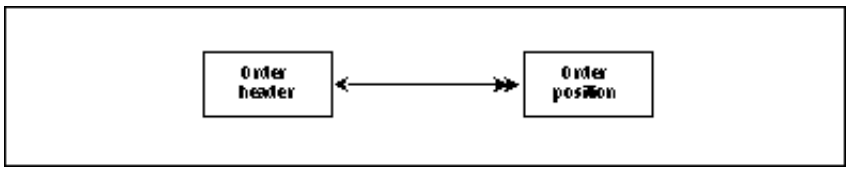
In an object view, all database operations, for example, Store, Update, etc. are implemented for an object on which they can be used.

The term "Object" refers to a group of business data which logically belongs together. An object corresponds, in its simplest form, to an entity identified during requirement analysis.

Complex objects can, however, represent a structure of entities which have relationships with each other.

A simple example of this is the object *Order* composed of the object *Orderheader* and the object *Order position*:

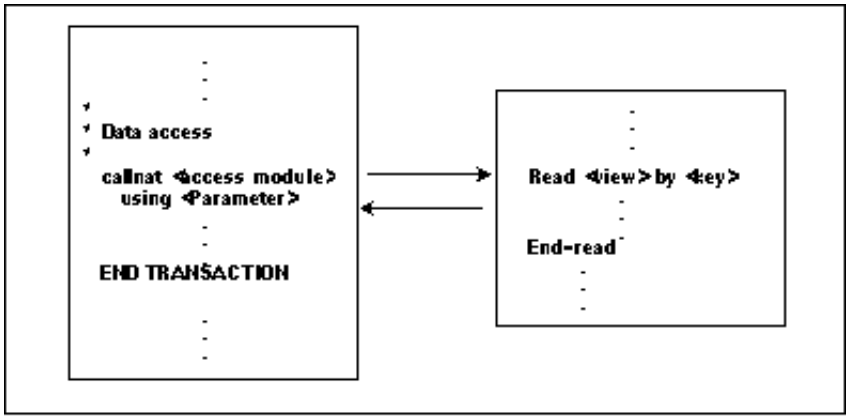
Order



An operation on an object is not implemented in a program directly as a database access. Rather, an access module associated with the object view is called with the appropriate parameters. The database access is then executed by the access module, and the results of this operation are returned to the executing program.

The activation and confirmation of data modifications is performed by an application subprogram (activation module). This module performs the transfer of preliminary data to the original data as well as the confirmation of the data update.

For further information, see section Transaction Logic.



The code of the activation module confirms successful modification with an END TRANSACTION statement or in case of an error condition, backs-out the transaction with the BACKOUT TRANSACTION statement. For distributed databases, it must be ensured that these statements are also applied to all relevant databases.

Besides database access, all necessary validations for data consistency and security are carried out by the access modules of the object views. This is done before any access to a database takes place. This is to ensure that these checks are always carried out regardless of the origin of the call.

Due to the different interfaces (with a single object only one object is passed back, while with multiple objects, several objects are passed back) used for processing single and/or multiple objects, at least two access modules are created for each object, i.e. one for single- and another for multiple-object processing.

Access modules are implemented as subprograms, and required data is supplied in parameter data areas.

Natural Objects Associated with an Object View

When an object view is created using the frame gallery, the following Natural objects are produced:

Object View Info

Contains information collected during the process of creating the object view, which is used later when creating dialogs which use the object view. The data is held in an internal format and can be viewed when the Info button is selected in the "Frame Gallery" dialog window.

Object View LDA

Contains the Natural view of the object and a start key definition used by the multiple object access module.

Constants LDA

Contains constants defining (1) the number of blocks of A100 used in the single object parameter data area; (2) the number of objects to be retrieved by each call to the multiple object access module; (3) a position number for each field in the object view local data area; (4) optionally, additional object-specific operation codes.

These are not automatically included but can be added manually. For further information, see Starting the Inplementation.

Single Object PDA

Contains the call interface for the single object access module.

Multiple Object PDA

Contains the call interface for the multiple object access module.

Preliminary Copies Copycode

Contains code for use by dialogs which use preliminary copies.

Single Object Subprogram

Encapsulates operations for accessing and modifying single objects.

Multiple Object Subprogram

Encapsulates retrieval operations for retrieving multiple objects.

Preliminary Copies Subprogram

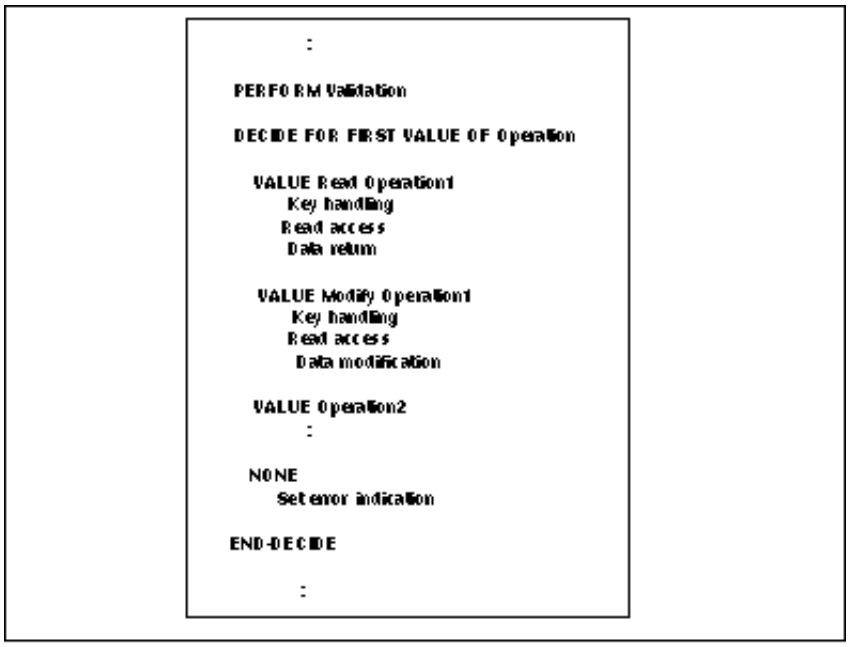
Transfers updates made to preliminary copies to the database.

Implementing Single-object Access

All database operations that read or change an object’s data are implemented in one of the access modules belonging to that object. This object data can be composed of fields from various DDMs, which together build a logical object. Before the execution of an operation in an access module, the required consistency checks are executed.

Access Module Structure

The structure of a single object access module is shown below:



The frame gallery produces a single-object access module which includes standard operations for READ, GET, STORE, UPDATE, DELETE and CHECK_EXISTENCE. Additional operations can be added manually using the subprogram editor.

Creation of Consistency Checks

The single-object access module produced by the frame gallery includes checks to ensure that parameters required by the standard operations are passed. Any further consistency checks and validations must be coded manually.

All required validations for data consistency checks are executed before the database modifications take effect.

These checks are collected in an inline subroutine similar to the following:

```
DECIDE ON EVERY VALUE OF Operation

   VALUE Operation1, Operation2, ...
      Consistency checks

   VALUE Operation1, Operation3, ...
      Consistency checks

   VALUE Operation5, ...
         :
         :

         :
         :
   NONE
     Set error indication

   END-DECIDE
```

Checks for multiple operations can be combined through the use of the statement DECIDE ON EVERY VALUE.

If the consistency checks contain database accesses to other objects (e.g. external key checks), then the access module calls an access module belonging to the other object's object view to carry out the database access. This procedure is necessary to allow data distribution.

To minimize communication overhead with distributed data storage, it may be desirable to perform certain checks during dialog processing. Nevertheless, these checks should also be included in the object view.

The checks do not, however, have to be coded more than once.

You can stipulate that only validation, but no modifying database accesses are executed by the access module, with the switch PZ_AS_VALID_ONLY.

It is also possible for certain field-specific checks, for example; check for numerical content, to be copied from the access module and directly coupled with the corresponding input field in for example the 'Change' event.

Application Program/Object View Interface

The interface between application programs and access modules contains various types of data:

- object independent data;
- object dependent data.

Object Independent Data

Parameters that are the same for every object are included here, and are defined in the parameter data area ZXAS000A. In particular, the following data is included:

- IN: Operation code (PZ_AS_OPERATION)
  The standard operation codes used by the application shell are contained in the local data area ZXA0000L. Additional operation codes can either be hard coded in the program or defined as constants in a local data area. Suggested code is included in the constants local data area, produced for an object view by the frame gallery.

- IN: Validation flag (PZ_AS_VALID_ONLY)
  Indicates that **only** validation is to be carried out.

- IN: Language position (PZ_AS_LANG_POS)
- IN/OUT: ISN of the record that will be or was read (PZ_AD_ISN)
- OUT: Error information
  - Response code (PZ_AS_RSP)
  - Existence of record. If the record exists the flag is set to 1 (PZ_AS_REC_EXIST)
  - Message number (PZ_AS_MSG_NUM)
  - Additional message information (PZ_AS_MSG_FILL)
  - Position number of the field in error in the view (PZ_AS_FLD_POS)
  - Index of the field in error for arrays (PZ_AS_FLD_OCC)
- OUT: Runtime information
  - Natural error code number ((PZ_AS_NAT_ERROR)
  - Natural program line in which error occurred (PZ_AS_NAT_LINE)
  - Natural object name (PZ_AS_NAT_PROG)

When using an SQL access module, an additional standard local data area ZXA000QL is required.

Object Dependent Data

For the single-object access module, three sets of object-dependent data are required and are produced by the frame gallery during object view creation:

- an local data area containing the Natural view.
- a single object parameter data area which includes a copy of the Natural view, as a group containing all or a subset of the fields of a complete view. It differs from the Natural view structure in that all structure definitions are omitted.

Only elementary fields are defined. Higher level fields are excluded.

This is done for the following reasons:

- The danger that errors will occur on parameter transfer is eliminated.
- Type conversion would otherwise not function properly with distributed data storage.
- Only essential fields are transferred.

This parameter data area is used both in the access module and in the application programs.

Additional object dependent data, for example constants for field positions and object specific operation codes are also defined in a local data area (the constants LDA).

Additional Object Dependent Data

Any additional object dependent fields and parameters should be defined in a separate parameter data area.

Error Handling

For error handling, the following applies:

- A position number is assigned as a constant for each field of the object view local data area. The necessary code is generated in the constants LDA.
- Each input field is assigned a field position within the dialog.
- If multiple views are affected, a view ID must be used to identify the view containing errors.
- If an error occurs in the access module, then the error number and the position number of the view field in error is returned. For arrays the indices for these marked errors are also returned.

If, for example, when deleting an object, additional reference checks are necessary, it is recommended to include in the error text the name of the object in which the object to be deleted is a foreign key.

Implementing Multiple-Object Access

Note:
This functionality is automatically generated.

In principle, all database operations which read more than one object record are implemented in an access module belonging to the object.

These "records" can be composed of fields from various DDMs, which together form a logical object.

The object view for multiple-object processing does not, as a rule, contain database accesses that change the data.

The access module returns a definable number of objects. The maximum number of supplied objects is defined in the index of the arrays of object fields found in the interface local data area and/or parameter data area of the application program.

If a block of records is returned, then the starting value of the next block to be read is also returned automatically. As a result, unlimited numbers of records can be read sequentially.

Because no accesses change data, the execution of consistency checks is not required.

Structure of the Multiple-Object Access Module

The structure of a multiple-object access module is as follows:

```
DECIDE FOR FIRST VALUE OF Operation

   VALUE Read Operation1
      Key handling
      Read access
      Data return

   VALUE Read Operation2
      Key handling
      Read access

   VALUE Read Operation3
                :
                :

                :
                :

   NONE
      Set error indication

END-DECIDE
```

The frame gallery produces a multiple-object access module which includes standard operation for LIST and GET_SET. Additional operations can be added manually using the subprogram editor.

Checking

In a multiple-object access module, checking is restricted to the required input parameters for control of the access module.

Application Program/Object View Interface

The area of communications between application programs and access modules contains various types of data:

- Object Independent Data
- Object Dependent Data

Object Independent Data

Parameters that are the same for every object are defined in the parameter data area ZXAM000A. The following is included:

- IN: Operation code (PZ_AM_OPERATION)

The standard operation codes used by the application shell are defined in the local data area ZXA0000L. Additional operation codes can either be hard coded in the program or defined as constants in a local data area. Suggested code is included in the constants LDA, produced for an object view by the frame gallery.

- IN: Number of required objects (PZ_AM_CNT_REC_NEEDED)
- IN/OUT: Number of objects found (PZ_AM_CNT_REC_FOUND)
- OUT: Status information
  - Response code (PZ_AM_RSP)
  - EOD flag, is set, when END OF DATA (PZ_AMEND_OF_DATA).
- OUT: Runtime information
  - Natural error code number ((PZ_AM_NAT_ERROR)
  - Natural program line in which error occurred (PZ_AS_NAT_LINE)
  - Natural object name (PZ_AM_NAT_PROG)

Object Dependent Data

For the multiple-object access module, two sets of object-dependent data are required and are produced by the frame gallery during object view creation:

- a local data area containing the Natural view (also used by the single-object access module). This local data area also includes a definition for a single start key.
- a multiple-object parameter data area which includes an array containing a copy of the Natural view, as a group which includes all or a subset of the fields in the Natural view. This definition differs from the Natural view structure in that all structure definitions are omitted. Only elementary fields are defined. Higher level fields are excluded.

A pair of start/thru values and a minimum value are defined for the selected search key for the object view. By default, these are used to read a range of records. For a composite key, you can include additional code to treat each component as containing independent start/thru values.

Additional start/thru values for further search keys can be added manually.

Additional selection criteria can also be defined in the parameter data area. With distributed data storage, it may be desirable to carry out all selections on the computer which holds the data.

This parameter data area is used as both in the access module as well as locally in application programs.

Error Handling

Errors will only occur if an invalid parameter (unknown operation code, invalid/ missing start value) or a call in the multiple-object access module to the single-object access module involving database updates is detected.

Error information passed by the single-object module to the multiple-object module must be passed on by the multiple-object module to the application program.

Error handling is performed in the same way as outlined above for the single-object access module.

Implementing Access to Preliminary Copies

The frame gallery produces two Natural modules to handle access to preliminary copies for an object view. A copycode contains the object type specific access to the preliminary records. An activation module transfers the preliminary data to the original data.

For further information on the use of preliminary copies, see Preliminary Copies.

Copycode for Access to Preliminary Copies

For access to preliminary copies, a copycode is created for each object. This copycode issued in the maintain dialogs and subdialogs, as well as in the corresponding activation module.

A modified version of this copycode is produced if the single-object parameter data area contains more than 4000 bytes.

Activation Module

The preliminary copies subprogram produced by the frame gallery can be modified as required.

If you do not want to use the predefined operation codes in the object view, you can specify the desired operation code using the following MOVE statement subsequent to the call of subroutine
Z_INIT_PARMS_ACCESS_ORIGINAL:
MOVE <operation-code> TO PZ_AS_OPERATION

Object View Implementation

Starting the Implementation

The object view which builds the access layer should be implemented as early as possible. The earliest possible point of time is upon completion of the database design.

Undoubtedly, not all operations can be implemented so early. Experience has shown that requests for additional operations arise during implementation. However, it should be possible to implement all standard operations.

The frame gallery automatically generates an object view which includes the actions READ, GET, STORE, UPDATE, DELETE, CHECK_EXISTENCE, LIST and GET_SET.

Object View Creation

Procedure

The procedure described here assumes that the access modules do not need to call access modules for further object views.

If this is not the case, see Object View Creation for Complex Objects.

Create the object view using the frame gallery (see Creating an Object View).

When object-specific operations are required complete the following:

Add all object-specific operation codes, which are individually valid for this object, and are not already in the standard local data area for operation codes to the local data area.

Assign initial values for operation codes in ascending order, beginning with 51 in the above mentioned local data area with the object-specific contents.

Implement the individual operations in the available suggested code DECIDE structure in the multiple- or single-object access modules.

Code the appropriate VALUE statement, and insert the essential coding.

For reasons of clarity, it is recommended that individual operations are placed after standard operations.

Test the object view.

Use:
for single-object access, the skeleton program ZXFAS00P
for multiple-object access, the skeleton program ZXFAM00P

If later, during the implementation phase, additional operations are required, then check that the operation codes have already been defined. If not, define the new operation codes, and implement them in the access modules.

Additional Information Concerning Multiple-object Access

With multiple-object access, three repositioning parameters can be used to delimit which records are to be read. These are:

- Start-value. Specifies the first record to be accessed (with the first read). When the access module has read as many objects as are to be returned by a single call to the module, it sets the start key to the next object to be read. When the access module is next called, it starts from this new start value.
- Thru-value. Specifies the last record to be accessed. Not set by dialogs produced using the frame gallery.
- Minimum-value. Not set by dialogs produced using the frame gallery.

A minimum value field is included in the multiple-object parameter data area but the access module does not include suggested code and the minimum key field is not used by dialogs generated using the frame gallery. You can add code to use the minimum value as follows.

The minimum value can be set to indicate that the components of a composite key are to be treated as independent search criteria. Before the first call to the access module, the minimum value is set to the start value.

For each key component, all objects with a value lower than that specified as the minimum value are ignored.

Example

Start-value:

Client ID	Customer ID	Retailer ID
BB	Carey	Bell

End-value:

Client ID	Customer ID	Retailer ID
DD	Zackery	McCalls

The first access will begin with the Start-value. The first 10 occurrences are placed in the parameter data area and the 11th occurrence is marked as the next start-value. If additional objects are required, the next access will begin with the 11th occurrence. This procedure is repeated until either the end-value is reached or no additional objects are required.

If the following Minimum-value is used, the number of objects returned is further reduced:

Client ID	Customer ID	Retailer ID
BB	Andrews	Bell

For client ID CA, for instance, only objects with a Customer ID greater or equal to Carey and a Retailer ID greater or equal to Bell will be returned.

Object View Creation for Complex Objects

To create object views for complex objects which contain other objects

1. Generate an object view for each individual object.
   Make sure that an object view for the lowest object in the hierarchy is implemented first.
2. In each access module that needs access to the lowest hierarchical object, code a call to the access module belonging to this object.

A hierarchical calling structure of access modules, matching the logical structure of the object, results.

Size Problem Solution

Size problems in an application program could be caused by the following:

- too many or too large data areas accessed in the program, resulting in a program which cannot be stowed, or
- total size of data areas is too large at run time due to nested calls to modules.

Because the object view concept suggests only one object view per object, for all fields in that object, problems of size can be solved by reducing the data areas in the application program.

There are two initial solutions:

Solution 1 - Data Areas in the Application Program too Large to Stow

If the data areas in the application program are too large, then the problem can be solved by reducing the size of the data areas in the application program.

To reduce the size of the data area in the application program

1. Encapsulate the call to the access module in an additional subprogram, which can be used as a type of "intermediate layer".
2. Define the interface between the application program and "intermediate module", so that only the data actually used by the application program is passed.

The intermediate module contains the "complete" interface for the object view.

Communication between the application program and the access module is no longer direct, instead the data in the intermediate layer is filtered. Through this method the number of object views for an object is reduced.

Solution 2 - Total Size of Data Areas at Runtime too Large

To reduce the size of the data areas at run time

- Create object views that include only interface fields which are actually used by the application program.

Take care in particular that not too many object views are created. Experience has shown that a "small" additional object view, containing the most frequently used data is sufficient.

This object view can be used in all application programs and/or access modules which only require the reduced data area.

Calling the Access Modules

Note:
The frame gallery automatically generates these calls.

Access modules are invoked in the following manner:

```
CALLNAT <Access Module Name>
   USING <Standard Parameter>
         <Object View Fields>
         <Additional Parameters> (optional)
```

Standard Parameters	For single-object modules: PZ_XAS000 For multiple-object modules: PZ_XAM000 This enables the control parameters for the access module to be transferred.
Object View Fields	Enables the group name of the view fields from the data area which contains the object fields of the view: **P_viewname**
Additional Parameters	These parameters contain all information which in addition to the standard parameters and the view fields must be passed to the access module, e.g. selection or sorting criteria. Using additional information is optional.

Error Handling

An error check should be made after calling the access module.

When using a single-object module, it is sufficient to check the variable PZ_AS_RSP.

The check for multiple-object modules is combined with a REPEAT loop.
This REPEAT loop is used when the number of database accesses makes a screen output necessary. It can also be used when the records returned by the access module require further filtering by the application program which must read call the access module more than once, e.g. to display a large number of objects in a list box.

The error check for the multiple-object module takes place within this REPEAT loop.

This call has the following structure:

```
REPEAT
   Call to the Access Module
   Error check
  END-REPEAT
```


Single-object Processing with the Multiple-object Module

If the application program already uses multiple-object access for an object, and single-object access is required, it is not necessary that a single-object object view, with accompanying data areas, be additionally implemented in the application program.

A single-object can also be read by a multiple-object module:

Set the parameter PZ_AM_CNT_REC_NEEDED to 1, and then activate the multiple-object module in the usual manner.

Due to the large size of the data area for a multiple-object module, this approach is only practical when multiple-object access is already used in the application program.

Note:
The subset of fields available with the multiple-object access module may differ from that available via the single-object access module, depending on how you created the object view using the frame gallery.

Reading Sequentially using the Multiple-object Module

Sometimes further filtering of the records, which are returned from the multiple-object module, is necessary in the application program.

If, this is the case, proceed as follows:

Code the secondary selection check inside the REPEAT loop to call the access module after the error enquiry.

Remove the rejected records from the transfer buffer, and move the remaining records together

Count the number of rejected records.

Subtract the number of rejected records from the value in the variable PZ_AM_CNT_REC_FOUND.

Repeat this loop process.

In this way, the rejected records are replaced. The start value for the next call to the access module is set by the access module during its previous call to the record following the last record it read.

Data Storage and Data Access

This section provides information and guidelines regarding data storage and data accesses.

Where relevant, alternative approaches are presented. All techniques can be combined with the use of the frame gallery production frames. Specific components or suggested codes which support a given alternative are also described.

The following topics are covered below:

- Terminology
- Concepts for Data Storage
- Time Stamped Data
- Histories
- Multiple Control
- Logical Deleting
- Multilingual Applications
- Access Paths
- Structuring Physical Files
- Synchronizing Competing Accesses

Terminology

Adabas C terminology is used throughout this chapter. The chart below displays the equivalent SQL terminology.

Adabas Terminology	SQL/Adabas D Terminology
File	Table
Record	Row
Data record	
Field	Column
Descriptor Index	Primary key
Super descriptor	Named Index (key with several parts)
READ	SELECT
READ(1)	SELECT SINGLE SELECT DIRECT

Concepts for Data Storage

Some business requirements influence not only the logic of the business functions but also the data storage.

To allow for these requirements, in certain circumstances, technical fields in addition to the fields for business data must be inserted into the entities.

The following sections describe some of the possible requirements, including the necessary modifications or extensions to the entities or files.

- Time Stamped Data
- Histories
- Multiple Control
- Logical Deleting
- Multilingual Applications

The implementation of the functions for processing these data is not described in this section.

Time Stamped Data

General

If there is a requirement to be able to process data for a key value dependent on particular periods, without modifying the data contents of other periods, then the data must be stored and processed in dated form.

Such data are only valid for a particular period. Since the defined periods for a key value cannot overlap, the whole data over all defined periods yields a life cycle of the "data record" for a key value.

Definition:

An entity is time stamped if, for each key value, several occurrences of a descriptive attribute can exist, where each occurrence is identifiable by a distinct period which does not overlap with another period. This guarantees that, at any point in time, only one value of a descriptive attribute exists.

Accordingly, a data record which contains an account status is not a time stamped data record, since the value of the account is only valid for exactly one point in time. The date, in this case, is a business content.

There are many models for utilizing the concept of time stamping for data storage and processing. The requirements that call for the use of a time stamping model can, in spite of their complexity, be summarized in a few words:

- The object must have a life cycle.
- The life cycle may or may not have gaps.
- Data from past or future periods is viewed from a given point in time, and can be modified or not.
- The limits of the periods are milliseconds, seconds, minutes, hours, days, and so on.

Time Stamping Concept Recommendation

The following time stamping concept is a recommendation for data storage of time stamped objects as well as access to the data from business functions.

The concept supports the following requirements:

- The data record has a life cycle which has either a defined end or can extend to "infinity".
- Gaps in the life cycle are allowed.
- Pre-time stamping, that is, processing the data for a period that at the time of processing is not yet valid, is allowed.
- Post-time stamping, that is, processing the data for a period that at the time of processing is no longer valid, can optionally be implemented.
- The limits of the periods are days, that is, the smallest possible definable period is one day.
- Frame gallery time stamping relates to the whole object, it is not field related.

If an entity contains time stamped and unstamped descriptions, then the division of the entity into one time stamped and one unstamped is to be considered.

Data Storage

- For each time stamped object, one physical data record is stored for each period in the database.
- The data structure is extended by two fields that contain the start and end dates of the period.
- Each period is identified through a combination of specialized key and start date.
- To simplify the read access, the time stamp values are stored in complementary form with its complementary value *"999999...9 - date"*. The exit time stamp must be in the format N12 of the Natural variable *DATX.

To prepare an entity for dating

1. Insert the following fields:
   xxx_EFD_INV (N12) effective-from-date (complementary)
   xxx_ETD_INV (N12) effective-to-date (complementary)
2. Define the superdescriptor for the access:
   xxx_KEY (...), consisting of
   xxx_specialized_key and
   xxx_EFD_INV
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:
   xxx_KEY_SEC (...), consisting of
   xxx_secondary key and
   xxx_EFD_INV

Access to Time Stamped Data

Data accesses are basically through a superdescriptor xxx_KEY or xxx_KEY_SEC.

The frame gallery philosophy to always encase data accesses in so-called access modules is also valid for time stamped data.

The suggested codes for implementing accesses to time stamped data are contained in the skeleton modules for creating access modules. The appropriate skeletons are identified by the suffix ’PE’, for "Period effective".

The following standard operations are typical:

Reading a Period

A physical data record is read using a key value and a date. The date is provided in complementary form. Using the value provided as from-date, the data record is read using a READ(1) (see following example).

Example:

For article number 4711, the following periods exist (represented in the form *YYYYMMDD*):

Art.Nr.	Valid from	Valid to
4711	19930101	19930330
4711	19930401	19930515
4711	19930516	19930831
4711	19930901	unlimited

Physically, the values are represented in the form *"9999..9 - date"*:

Art.Nr.	Valid from	Valid to
4711	80069898	80069696
4711	80069598	80069484
4711	80069483	80069168
4711	80060098	00000000

The date is *02/05/93* with the complementary value *80069497*. Using this date, the start date is accessed. By using the access READ(1), the next highest value is read.

Result of read access: *80069598* - which is the equivalent of *01/04/93*.
With this, the validity period is identified. The date exists in the period *01/04/93 to 15/05/93*.

Existence Checking on Key Value

A read access READ(1) with key value and date "infinity", complementary value *"0"*, is used. This results in a record found, if one exists with the specified value.

Adding a New Key Including a Period

The data record is stored with key value including start and end date in complementary form.

Modifying the Data of a Period

The descriptive attributes of the data record are modified, without modifying start or end date.

Modifying the Limits of a Period

Depending on the kind of modification and the specialized requirements, quite varied activities can be necessary:

- **Overlapped Periods.** All periods except the modified period itself, which are completely overlapped by the modification of a period are deleted.
- **Partly Overlapped Periods.** Already existing periods which are partially overlapped as a result of a time period modification are automatically shortened.

The existing period which is impacted by new Start Date of the modified period is:

- End date of old period = start date of the modified period minus 1 day.
- New end date of the impacted period is prior to its current end date.

The existing period impacted by the new End Date of the modified period is:

- Start date of the old period = end date of the modified period plus 1 day.
- New start date of the impacted period is **after** its current start date.

Fixing the New Limits

The limits of the period being processed are adjusted correspondingly.

Example:

Existing period:

1 01st May - 31st May

2 01st June - 15th June

3 16th June - 17th August

4 18th August - 20th September

Period to be modified: 01st June - 15th June

New limits:	15th May - 01st September		
To delete:	16th June - 17th August		
To modify:	01st May - 31st May	->	1st May - 14th May (new end date)
	18th Aug. - 20th Sept.	->	2nd Sept. - 20th Sept. (new start date)

Result:

1 01st May - 14th May .

2 15th May - 01st Sept.

3 02nd Sept. - 20th Sept.

Adding a New Period for a Key

- **Overlapped Periods.** All periods completely overlapped by the new period are deleted.
- **Partly Overlapped Periods.** All partly overlapped periods are shortened, as described in the section: Modifying the limits of a period.

- **Adding a new period.** A new data record, with the limits given, is added.

Deleting a Period

- **Overlapped periods.** All periods that are completely overlapped by the deleted period are deleted.
- **Partly Overlapped Periods**. All partly overlapped periods are shortened, as described in the section: Modifying the limits of a period.

Example:

Existing periods:

1	01st May - 31st May
2	01st June - 15th June
3	16th June- 17th August
4	18th August - 20th September

Period to be deleted: 15th May - 01st September

To be deleted:	01st June - 15th June 16th June - 17th August	
To be modified:	01st May - 31st May 18th Aug. - 20th Sept.	-> 01st May - 14th May -> 02nd Sept. - 20th Sept.
Result:	01st May - 14th May 02nd Sept. - 20th Sept.	

Deleting a period

The data record is deleted.

Deleting a key

All data records belonging to a specialized key are deleted. For this, a READ access is used with start value *key-value* and *start-date equals "infinity",complementary-value ’0’*. This results in all affected data records to be found.

Histories

Histories are used when every data modification must be logged.

With every data modification, the original data record is retained. It is merely identified as historical. The modified data record is recorded as a copy of the original record including the modification as a "valid" record in the data content. Every historical record must contain the time of the modification, to be able to find out the chronological order of the modifications.

The concept of keeping histories can be implemented in various ways. The use of this concept affects the data modelling and also the implementing of the specialized functions affected.

In the remainder of this section, three ways of keeping histories are described:

- Histories in the original file with validity identifier.
- Histories in the original file with additional key.
- History keeping in a separate file

Histories in the Original File with Validity Identifier

In this way, all data records that have become historical by modification are kept in the original file. To separate the historical data from the current, a validity identifier is introduced.

Access to the current data is only through the combination of validity identifier and specialized key. Access to historical data is through a separate key.

Data Storage

For logging the modification time, the entity is extended by a field which is used to enter the time stamp of the modification time. For this, the format N20 is provided. To guarantee the uniqueness of the time stamp, the value of the Natural system variable *TIMESTMP (B8) is used as modification time. It can be completely portrayed in N20.

If the need exists to read all historical data in a chronological order independent of the specialized key, the time stamp must be defined as key value.

Additionally, a validity identifier is introduced and a combined key is constructed from validity identifier and specialized key.

Finally, a history key for access to the historical data is built as a combination of specialized key and time stamp.

To prepare an entity for history keeping

1. Insert the following fields:
   xxx_PTS_HIST (N20) time stamp of modification (Processing Time Stamp)
   xxx_ACTIVE (N01) validity identifier (Active Flag)
   Important:
   The validity identifier must, in any case, be added with null-value suppression.
2. Define the superdescriptor for the access to current data:
   xxx_KEY (...) consisting of:
   xxx_ACTIVE and
   xxx_specialized_key
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:
   xxx_KEY_SEC (...) consists of:
   xxx_ACTIVE and
   xxx_secondary_key
4. Define the superdescriptor for the access to historical data:
   xxx_KEY_HIST (...) consisting of:
   xxx_specialized_key and
   xxx_PTS_HIST
5. If the entity secondary key must be read through the historical data, define for each secondary key a further superdescriptor in the form:
   xxx_KEY_SEC_HIST (...) consisting of:
   xxx_secondary_key and
   xxx_PTS_HIST

Accesses

- **Reading a valid data record**
  Access is through the superdescriptor xxx_KEY with start value specialized key and validity identifier 1 (historical records have validity identifier 0).
- **Existence checking of a valid data record**
  According to specialized requirements, an unsuccessful read access to valid data must follow an access to historical data, to find out whether a key was already used at some time in the past.
- **Adding a valid data record**
  The record is stored with validity identifier 1.
- **Modifying a valid data record**
  The modifications are on a **copy** of the original data record. The copy can be stored in a temporary store or as historical record in the original file (then access to the copy must be through the time stamp).
  After finishing the modifications, the original record is **copied** and provided with validity identifier 0 and the current time stamp. Then the modifications are brought into the old original record. All finishing activities are completed within one database transaction.
- **Deleting a valid data record**
  Deleting data records is, as a rule, not usual when using a histories concept. Should a deletion still be required, the validity identifier of the original record is merely set to 0 and the current time stamp entered.
- **Reading a historical data record**
  Access is through the superdescriptor xxx_KEY_HIST with start value specialized key and required time stamp.
- **Adding a historical data record**
  See section Modifying a valid data record.
- **Modifying a historical data record**
  Modification of historical records is, as a rule, not usual.
- **Deleting a historical data record**
  The physical deletion of individual historical data records is not usual. Historical data are, as a rule, only removed from the data content in the course of archiving.
  Access can then, according to requirement, be either through the time stamp xxx_PTS_HIST or through the

history key xxx_KEY_HIST.
Usually, at a particular point in time, all historical data records that are older than a particular date are archived. For this, the time stamp xxx_PTS_HIST is used.

Histories in the Original File with Additional Key

In this way, all data records that have become historical by modification are kept in the original file. To separate the historical data from the current, an additional key for the historical specialized key is introduced.

Accesses to current data are through the valid specialized key. Access to historical data is through the separate historical specialist key.

Data Storage

For logging the modification time, the entity is extended by a field which is used to enter the time stamp of the modification time. For this, the format N20 is provided. To guarantee the uniqueness of the time stamp, the value of the Natural system variable *TIMESTMP (B8) is used as modification time. It can be completely portrayed in N20.

If the need exists to read all historical data in a chronological order independent of the specialized key, the time stamp must be defined as key value.

Moreover, an additional field is inserted which matches the specialized key in format and length.

Finally, a history key for access to the historical data is built as a combination of specialized key and time stamp.

To prepare an entity for history keeping

1. Insert the following fields:
   xxx_PTS_HIST (N20) time stamp of modification (Processing Time Stamp)
   xxx_ID_HIST (...) historical specialized key (Format and length = specialized key)
2. If the entity contains secondary keys through which there must be access, define for each secondary key a further field in the form:
   xxx_ID_SEC_HIST (...) historical secondary key (Format and length = secondary key)
3. Define the superdescriptor for the access to historical data:
   xxx_KEY_HIST (...) consisting of:
   xxx_ID_HIST and
   xxx_PTS_HIST
4. Define the superdescriptor for the secondary keys:
   xxx_KEY_HIST_SEC (...) consisting of:
   xxx_ID_HIST_SEC and
   xxx_PTS_HIST

Accesses

Accesses through the specialized keys are not affected.

All accesses to historical data are through the key xxx_KEY_HIST or xxx_KEY_SEC_HIST.

Remarks on these accesses are to be taken from the previous section.

History Keeping in a Separate File

This approach results in a complete separation of the valid from the historical data. All historical data are kept in a separate entity.

This construct presupposes no technical fields in the file for the original data. The history file is an image of the original file, where additionally, for the purpose of chronologically sorting the historical data records, a time stamp is maintained which indicates the time of the history.

Data Storage

To store the historical data, a separate entity is added, which has exactly the same structure as the original.

Additionally, a time stamp is added as field, to document the modification time of a data record. This time stamp is defined as a descriptor or, as required, as part of a superdescriptor from specialized key and time stamp.

To prepare the data structures for history keeping

1. Add a file that has the same field structure as the original file.
   According to requirement, descriptor definitions can be dropped.
2. Insert the following field:
   xxx_PTS_HIST (N20) time stamp of modification (Processing Time Stamp)
3. Define this field as descriptor if there is a need to read all data of the file in chronological order.
4. Define the superdescriptor for the access to historical data:
   xxx_KEY_HIST (...) consisting of
   xxx_specialized_key and
   xxx_PTS_HIST
5. If the entity contains secondary keys through which the historical data must be read, define for each secondary key a further superdescriptor in the form:
   xxx_KEY_SEC_HIST (...) consisting of:
   xxx_secondary_key and
   xxx_PTS_HIST

Accesses

- **Reading a valid data record**
  Not affected.
- **Existence checking of a valid data record**
  According to specialized requirements, an unsuccessful read access to valid data must follow an access to historical data, to find out whether a key was already used at some time in the past.
- **Adding a valid data record**
  Not affected.
- **Modifying a valid data record**
  The modifications are on a temporary copy of the original data record.
  After the modifications are finished, the contents of the original record including the current time stamp are transferred into the history file. Then the modifications are brought into the original record.
  All further activities are completed within one database transaction.
- **Deleting a valid data record**
  The deletion of data records is, as a rule, not usual when using a history concept.
  If a deletion is still required, then a copy of the original with the current time stamp is transferred into the history file and the original is deleted.
  Access to historical data is analogous to the method used in the previous section, however, with the difference that another file or user view is accessed.

Multiple Control

From a business area perspective, certain data modifications must be confirmed before being used.

The number of necessary confirmations depends upon specialized requirements. Usually, one additional confirmation is enough (**double-review principle**).

In rare cases, the **triple-review principle** is used. For this, 2 additional confirmations are necessary.

Data modification and each confirmation are, as a rule, carried out by different people.

The use of this concept must be considered both with the design of the entity and also with the function structure.

There are various ways of implementing this concept. The spectrum ranges from very simple to very complex.

Complex Variant

The following complex variant, which is user friendly, is not however described below in general terms:

- There is an entity that contains all information about the processing state of all affected data records. Here, not only are the current states recorded, but also the entire processing history is noted, with specialized key, status, processor and time of processing.
- There is a further entity, that contains all status info that are necessary for a specialized entity.
- Finally, there is an entity that contains information about which user can process which data record in which status and, if the data record must be further processed, to whom appropriate information must go.

For all entities, appropriate maintenance functions must be generated.

With this variant, the specialized entities remain untouched. Before each access to a specialized data record, there are accesses to the status files.

Simple Variant

A very simple variant would be the following, which is described in detail:

- To each specialized data record, a status field is allocated.
- Additional fields for logging the last processor and the date of processing can be added.

Data Storage

The state of a data record is documented with the help of a status field. The status field contains information about the progress of processing, such as, *"added","checked once", "approved"* and *"declined"*.

As required, additional data fields are to be provided for the user who carries out an activity and the time that the activity is carried out.

To prepare an entity for the multiple review principle

1. Insert the following fields:
   xxx_STATUS (A01) status field
   xxx_USER (A08) last processor
   xxx_PTS (N20) time stamp of last processing
2. Define the superdescriptor for access to the data in dependence on its status:
   xxx_KEY_STAT (...) consisting of:
   xxx_STATUS and
   xxx_specialized_key
3. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:

xxx_KEY_SEC_STAT (...) consisting of:
xxx_STATUS and
xxx_secondary_key

Accesses

All accesses to the entity are through a superdescriptor

xxx_KEY_STAT or
xxx_KEY_SEC_STAT

The remaining processing, such as, modifying or deleting, is dependent on the specialized requirements.

Logical Deleting

In order that data previously deleted can be recreated, it must not be physically deleted, but only logically deleted.

Various solutions are possible. With the most comprehensive, the data are provided with a history, where the data only have a history in the case of a logical deletion. The data structure here matches that for complete history.

In the following, a simpler method is introduced. Here only the last content of the data to be logically deleted is stored. In the case of a recreation of the data, the data record is simply "activated" again.

Data Storage

The data structure is extended by a validity identifier. For access to valid data records, again a superdescriptor is defined. Access to deleted data is through the specialized key.

To prepare an entity for the logical deleting

1. Insert the following field:
   xxx_ACTIVE (N01) validity identifier (Active Flag)
2. Define xxx_ACTIVE null-value suppressed.
   The field will contain 0 when the record is logically deleted.
3. Define the superdescriptor for access to the valid data:
   xxx_KEY (...) consisting of:
   xxx_ACTIVE and
   xxx_specialized_key
4. If the entity contains secondary keys, define for each secondary key a further superdescriptor in the form:
   xxx_KEY_SEC (...) consisting of:
   xxx_ACTIVE and
   xxx_secondary_key

Accesses

All accesses to valid key values are through a superdescriptor xxx_KEY or xxx_SEC_KEY. Accesses to logically deleted data are through the specialized key.

- **Reading a valid data record**
  Access is through the superdescriptor xxx_KEY with start value specialized key and validity identifier 1 (logically deleted records have validity identifier 0).
- **Existence checking of a valid data record**
  According to specialized requirements, an unsuccessful read access to valid data must follow an access to logically deleted data, to find out whether a key was already used at some time in the past.
- **Adding a valid data record**
  The record is stored with validity identifier 1.
- **Modifying a valid data record**
  Not affected.
- **Logically deleting a valid data record**
  The active identifier is set to 0.

- **Physically deleting a valid data record**
  Deleting data records is, as a rule, not usual when using logical deletion. Should a deletion still be required, reading and deleting is then through the specialized key.
- **Reading a logically deleted data record**
  Access is through the specialized key. Through this, a logically deleted or a valid data record can be found. The identification of a logically deleted record is through further selection: xxx_ACTIVE must be 1.
- **Adding a logically deleted data record**
  See section Deleting a valid data record.
- **Modifying a logically deleted data record**
  Not usual.
- **Deleting a logically deleted data record**
  Deleting logically deleted data records is, as a rule, not usual. This deleting is, in general, only done with physical deletion of a valid data record.
- **Recreating a logically deleted data record**
  The xxx_ACTIVE identifier is set to 1.

Multilingual Applications

Many applications must be available in multiple languages. For this, first the interface (dialogs and command language) must be prepared in multiple languages. Also on the data level, entities will be identified that contain multilingual elements.

Language-dependent data elements of an entity are frequently identified in the design phase. As a first step, it must be decided whether the multilingual fields are to be left in the data structure or whether a separate entity that contains the multilingual fields will be generated.

Using a Separate Entity

The new entity contains all language-dependent fields of the data structure. In addition, a language field is inserted.

Use this variant only when a doubling of the number of read accesses when accessing language-dependent fields is justifiable.

Data Storage

To define the structure of the language-dependent entity

1. Define per language-dependent field a field of the same format and same length as the original:
   xxx_Field (A..)
2. Define all key values that are used to read language-dependent data in the language-dependent entity:
   xxx_ID (...)
3. Define additionally a field with the form:
   xxx_LANG (N01) language code of the language considered
   Take the language codes from the frame gallery table *Language*. Use the content of the field *xxx_ID*.
4. Define per key value of the original entity a superdescriptor of the form:
   xxx_KEY consisting of xxx_LANG and xxx_ID
   In rare cases, it can be sensible to turn round the fields of the superdescriptor and so have the key value in front.

Accesses

All accesses to the specialized object are performed by two real accesses:

- Access to the language-**in**dependent data,
- Access to the language-dependent data.

Accesses to the language-dependent data are always through a key of the form xxx_KEY.

Language-Dependent Fields in the Entity

With this variant, language-dependent and language-independent fields are held in one data structure. Naturally such structures require a special design.

Data Storage

There are numerous possibilities for portraying language-dependent fields. The "right" design can, however, only be selected depending on the context.

In the following, only the variants "multiple field" and "periodic group" are represented.

- **Multiple Field**
  The storage of language-dependent data as multiple field is recommended when there is a small number of multiple-language fields per record and when the field must be a descriptor or part of a superdescriptor through which there must be sequential reading.
  One occurrence of a multiple field is used per language.
  To enable direct access to a particular occurrence of the multiple field, a byte (N1) is placed before the actual field content.
  The byte contains the frame gallery language position of the language to which the field content belongs.
  The multiple field is therefore one place longer than the original field:
  Define then per language, instead of the language-dependent field, a multiple field:
  xxx_LFIELD(A..) = language_position (N1) + field(A..)
  When defining the structure, take note of the maximum possible physical record length. It must not be exceeded.

- **Periodic group**
  The portrayal of language-dependent fields in a periodic group can only be used when none of the fields is used as a key value or part key.
  A language-dependent field content is put in the occurrences of the periodic group that matches the frame gallery language position (LZ_LANG_POS in LDA ZXXGLOBL).
  Define then a periodic group that contains all language-dependent fields that are not used as key or part key:
  xxx_LANG_FIELDS (9)
  xxx_FIELD 1 (A..)
  :
  xxx_FIELD n (A..)
  With frame gallery, 9 languages are available.
  When defining the structure, take note of the maximum possible physical record length. It must not be exceeded.

Accesses

When you are reading through a key value, descriptive language-dependent data can be taken without problem from the occurrence that matches the frame gallery language position.

Reading accesses are only possible when using multiple fields.

In the following, only access to multiple fields is considered:

- **Taking out language-dependent attributes**
  The language-dependent field content must always be taken through an intermediate field that is redefined analogous to the data field into language and original field.

- **Reading through a language-dependent key**
  Access is through the multiple field xxx_ID with required language position and specialized key.

- **Sequential read access to language-dependent key**
  When sequentially reading through multiple fields, there is generally the problem of being able to immediately identify the last data record of a language. Therefore special break-off conditions must be formulated within the read loop:
  xxx_FIELD (language) LT Start_value
  xxx_FIELD (language) LE #OLD_xxx_FIELD
  If no start value was given, the first value in the language should be established by a HISTOGRAM statement with an end condition.
  #OLD_xxx_Field Value last time through read loop
- **Adding a language-dependent field**
  When first adding the record that contains the multi-language field, the occurrences for all other languages defined in the system are also to be added, to avoid the multiple fields pushing together and the consequent destruction of the direct access path. For this, only the language position is filled and the actual field content stays blank.

Access Paths

In addition to the access paths using a descriptor or superdescriptor, there are accesses that require certain preconditions to lead to a result simply and with good performance.

These access paths are described below.

- Sequential Reading through Nonunique Key
- Upper/Lower Case

Sequential Reading through Nonunique Key

Nonunique key fields, usually names, can cause problems in the paging logic.

To avoid this problem when reading, all keys that are not unique by nature are made unique with the help of additional fields.

For a nonunique key, a superdescriptor is defined which consists of the nonunique key and one of this data record's unique identifying fields (for example, the identifying key).

The nonunique key must not always be taken completely into the superdescriptor. As a rule, the first 15 to 20 characters are enough.

To extend the entity

- Define for each nonunique secondary key a superdescriptor:
  xxx_KEY_SEC unique secondary key consisting of:
  xxx_NAME (1-10) bytes 1-10 of the name
  xxx_ID specialized key
  When no nonsequential accesses are required, the nonunique key need not be defined as a descriptor. All sequential accesses are then through the superdescriptor.

Upper/Lower Case

Frequently, alphanumeric values, for example, names, should be included on the dialog and also displayed in upper and lower case.

If this is a key value, through which there must be access, storing in upper and lower case can have the danger that the value sought may not be found because, for example, during entry an upper case letter was accidentally entered in the middle of the key.

This problem can be avoided as follows:

- The field value is stored twice, once in upper and lower case and once only in upper case.
- As key for reading, the value stored in upper case is used, the value in upper and lower case is only used for display.

This procedure requires the insertion of an additional field into the entity.

To extend the entity

- Define for each key value that must be displayed in upper and lower case an additional field.
  xxx_CODE_UC specialized key in upper case
  This field is added as descriptor. The field that contains the value in upper and lower case is defined as descriptive attribute.

Read accesses are always through xxx_ID_UC. Modifications of the field content must be made in both fields.

Structuring Physical Files

Note:
This section can be skipped if you use SQL since the SQL database design uses a "flat" file structure.

When creating Adabas C files, there are the following possibilities:

- translating one entity from the requirements analysis 1:1 into one physical file,
- defining several entities in one physical file.
  The entities are arranged one behind another in the physical file.

The preferred method for an entity can be determined from the context.

For example:

- number of data elements,
- expected data volume,
- expected access and modification frequency,
- logical associations of the entities.

An entity with a large number of data elements, high expected data volume, and extremely high likelihood of modification is certainly to be represented as its own physical file.

On the contrary, several smaller entities can be put together in one physical file to reduce the number of physical files.

The following rules should be observed:

- The entities are placed one behind another in the file. At the beginning of a new entity, a corresponding comment is inserted in the PREDICT file description.
- Mixing of data elements from different entities is not allowed.
- One field is basically not used by two different DDMs, to avoid influence of the DB designs on the program logic.
  This is also not done if the field content is the same. A field per DDM is always added, see the following example:
  physical file

  *** * View ABCD ***

  ABCD_ID
  ABCD_NAME ! DDM for ABCD
  ABCD_FIELD1 !

  *** * View EFDH ***

  EFGH_ID
  EFGH_NAME ! DDM for EFGH
  EFGH_FIELD1

- READ physical or READ BY ISN on the DDMs of such a file is to be avoided, since the ISNs of the other DDM of this file would also be read.

Synchronizing Competing Accesses

The following topics are covered below:

- General
- Use of Locking Concepts
- Pessimistic Locking Concept
- Optimistic Locking Concept
- Organizational Locking Concept
- Processing Without Locking Concept

General

Concepts for synchronizing competing accesses are, in general, needed when the possibility exists that data records could be modified at the same time by several users or programs.

Various concepts can be used to avoid this problem.

These concepts extend from purely organizational measures, through checking for data modifications during the processing time to active locking of data records, as soon as the data records to be processed are identified.

A selection of these concepts is described below. All concepts can be used when implementing specialized functions on the basis of the frame gallery production frames.

Use of Locking Concepts

Locking concepts are used in the case of competing updating. Display functions usually do not require the use of a locking concept.

Use a locking concept, if:

- several users per dialog function could modify the same data records at the same time;
- parallel with the dialog service, batch programs are running that modify data;
- evaluations are being made that require a defined state of the basic data;
- data that serve as a basis for the current processing of data but must not be modified during the processing.

Important: Decide on one locking concept per entity.

The use of various locking concepts on one entity will lead to difficulties with the implementing of the specialized function or even lead to data inconsistencies.

More reasonable for implementing and maintenance of the application is the use of one locking concept for all data that are processed in the application.

Pessimistic Locking Concept

Concept

As soon as a data record is identified for processing, a lock is written for this data record. This lock remains until the close of the logical transaction. In this time, no other user can modify the data record.

This Naturally presupposes that, after the identification of a key value, there will first be a check whether the data record is already reserved for another user. If this is the case, then access to the data record is rejected.

The pessimistic locking concept is supported by the frame gallery production frames and suggested codes. If, when implementing, you do not depart from the suggested codes, pessimistic locking is automatically implemented.

The pessimistic locking concept is the "safest" and, for the end user the most comfortable of the concepts represented here.

Operation

From the viewpoint of pessimistic locking, a transaction looks as follows:

1. identification of the data record.
2. checking whether a lock already exists.
   - if yes, the access is rejected.
   - if no, a lock is written.
3. data modification.
4. transaction end: the lock is removed.

For each further data record that is identified for modification during the running transaction, a lock can, if necessary, be written. All locks are removed at the end of the logical transaction.

Data Storage

No particular measures are necessary.

Determining the Data to be Locked

Besides individual data records, key areas or entire objects can be locked. This is part of the frame gallery basic functionality.

Note:
When using frame gallery, the primary key is locked. Any additional keys must be manually locked with subroutine Z_LOCK_RECORD.

Which data key in which specialized function must be locked, must be decided from the specialized context.

Frequently, it is not necessary to lock every individual data key that is processed in a specialized function:

- Entire objects should be locked when the majority of the data records in a specialized function is modified or else must not be modified during the running transaction, as, for example, happens from time to time in batch programs.
- Key ranges should be locked when, at the beginning of the transaction, it is already clear that precisely these ranges are affected by the processing.
- Individual keys should be locked when the affected key values can only be identified one after another.
- Keys should not be locked when they are stored in a tree-like structure, whose root object is already locked and there is no possibility of otherwise accessing these hierarchically subordinate data keys.

Application

This approach guarantees that even very time-intensive logical transactions can be successfully closed. Once a transaction has begun, the data are reserved until the end of the transaction.

The approach is therefore especially suitable for dialog systems, since even specialized functions, which run through a large number of dialogs can not be destroyed during processing by other users or transactions.

Even in parallel use of dialog and batch functions, the approach is very suitable. The two types of functions cannot conflict with one another.

Optimistic Locking Concept

Concept

Note:
Frame gallery does not generate optimistic locking concept. It must be manually implemented.

Every data record contains a time stamp that documents the time of the last modification.

As soon as a data record is identified for .processing, the time stamp of the original record is saved in a temporary store, in general in the GDA.

At the close of the logical transaction, before the transfer of data, there is a check whether the time stamp in the original record has been modified.

If the time stamp in the original record was **not** modified, the transaction is closed in an orderly way.

If the data record, however, was modified, data transfer is terminated and the modifications of this transaction are lost.

The optimistic locking concept can be applied when using the frame gallery production frames. However, there must be a departure from the suggested code.

In contrast to the pessimistic locking concept, the optimistic locking concept is less comfortable, since not until the close of the transaction is there a check whether the modifications can be transferred into the database.

Operation

From the viewpoint of optimistic locking, a transaction looks as follows:

1. identification of the data record.
2. saving of the time stamp.
3. data modification.
4. transaction end: the time stamp is the same.
   - if yes, the transaction is closed in an orderly way.
   - if no, the transaction is stopped. The modifications are lost.

For every further data record that is identified for modification during the running transaction, a time stamp must be saved, which at the end of the transaction must be checked.

Data Storage

Every data record must contain a time stamp. Since this time stamp must be unique, it is provided with the converted content of the Natural system variable *TIMESTMP.

To extend your entity

xxx_PTS (N20) Modification time stamp

Determining the Data to be Locked

Which data key in which specialized function must be locked must be decided from the specialized context.

Since the locking runs through a time-stamp comparison, the time stamp of every affected data record must be checked.

The single exception is hierarchically structured data records. If a data record is stored in a tree-like structure, whose root object is already locked and there is no possibility of otherwise accessing this hierarchically subordinate data key, no time-stamp comparison must take place.

Application

This approach can lead to data that are modified in a logical transaction not being taken over into the data content, because the original data meanwhile were modified by another logical transaction.

The approach is therefore not suitable for dialog systems that also contain specialized functions that run through a large number of dialogs, since the likelihood of data being modified during the transaction is great.

It is suitable when the probability of competing accesses is small. This is the case when the logical transactions can be closed very quickly and few transactions access the same data.

Organizational Locking Concept

Concept

This procedure presupposes that the possibility of competing accesses is hindered by organizational measures.

This can, for example, be guaranteed when only one user is responsible for the processing of particular key ranges. This should then, however, be guaranteed by measures for protecting field content.

The organizational locking concept can be applied when using the frame gallery production frames. Here you must, however, deviate from the suggested code.

This concept pursues a completely different approach from the concepts previously described. At data access there is no effort in locking. The effort, however, appears at another place.

Data Storage

No measures are necessary.

Application

This concept can only be used in exceptional cases, since normal dialog systems are constructed to make the data available to multiple users.

The application of this approach is only useful for very small systems, which are only used by very few users.

Processing Without Locking Concept

This procedure is not advisable for multi-user systems, since the dialog processing can be disturbed by competing accesses, deadlocks, mutual overwriting, and so on.

Transaction Logic

The term transaction logic is used to describe all activities which are executed from the beginning to the end of a logical transaction.

A logical transaction also comprises all data modifications which have been confirmed as a unit by the end user.

Logical transactions are supported by the frames for main dialogs, subdialogs, and modal windows. The start and end of transactions takes place in a main dialog.

The following topics are covered below:

- Transaction Logic of the Modification Functions
- Cancelling a Transaction

Transaction Logic of the Modification Functions

The frame gallery transaction logic guarantees that data modifications resulting from a single transaction are either all applied or are not at all applied to the database.

The user receives a personal copy of the data records to be modified. This copy is identified using a user identifier and a unique time stamp for the transaction.

All modifications are initially applied to the preliminary copies only. Only at the termination of the transaction are the modifications applied to the original database.

Cancelling a Transaction

In that data modifications are only applied to preliminary files of the original data, all data modifications can easily be removed by simply deleting the preliminary files and any possible locking markers.

Data Transfer

Complex functions usually consist of several dialogs. In each of the dialogs, a section of the business functions is processed. The data transfer between these dialogs can take place in the following ways:

- Between all dialogs created with frames, data can be transferred via parameter PZ_DATA. For modal windows this method is included in the suggested code.
- Business data can be transferred between the main dialog and subdialogs via preliminary records.

Preliminary Copies

When starting a transaction, the original data are read from the database and the user is provided with a copy, the preliminary copy. Modifications to the preliminary copy can only be viewed from the respective function. This means that other users have access to the original data, but not to the preliminary copy.

Data modifications in dialogs are transferred to the preliminary copies using the command Z_CONFIRM. Command Z_REFRESH reverts the data to the state after the most recent modifications were confirmed.

The preliminary copies can be stored either in the preliminary file Z_PRELIMINARY or in main storage. This can be determined for each function in the respective application shell’s function maintenance. In the main memory, a maximum of 18KB of data can be stored per transaction. For larger quantities of data, the preliminary file must be used.

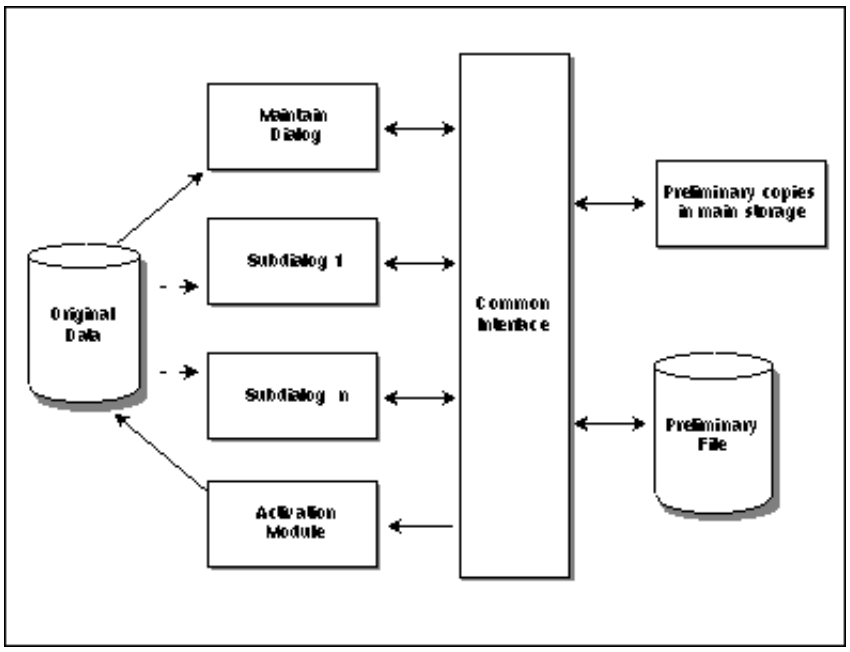
Access to the preliminary copy for an object view is embedded in a copycode. This copycode is used by the activation module and the dialogs in subroutine view_ACCESS_PREL. Access to the preliminary copy is transparent, irregardless of whether the data are stored in the main memory or in the preliminary file.

When a subdialog is opened, the preliminary copies are read first. When the modifications are confirmed in the subdialog, the preliminary copies are updated.

When a transaction is closed with command Z_SAVE, the data in the preliminary copy are transferred to the original data. Therefore, the activation module is used (as part of the object view).

The activation module can optionally be installed on a server. This is useful in the case of complex transactions.

In this case, the respective flag must be set in the application shell’s function maintenance. The actual RPC occurs via a standard activation module. This module transfers data to the server and calls the business activation module.


Maintain Dialog
Subdialog 1
Subdialog n
Activation Module
Original Data
Common Interface
Preliminary copies in main storage
Preliminary File

Locking Logic

The frame gallery suggested code uses the pessimistic locking concept. This involves the following procedures:

- Lock Marker Check and Write
- Remove Lock Markers

Lock Marker Check and Write

Immediately following the identification of a key value, it is checked whether or not a lock marker for the key ID of the associated object has been written. If yes, the access to the object is denied. Otherwise, a lock marker is written.

The locking of data occurs in a standard fashion during the start of the transaction. In the custom component Z_LOCK_RECORD the frames for main dialogs are assigned the object identifier and the key ID. The standard subroutine Z_CHECK_AND_LOCK_RECORD assumes responsibility for locking the data record.

Furthermore, it can be necessary to lock additional records at a given point during transaction processing. For this purpose, the suggested code of customizable component Z_LOCK_RECORD can be copied and adapted accordingly for use within any other component.

Remove Lock Markers

Lock markers are removed when the transaction is terminated normally or is cancelled. The removal is performed by the dialog frame. The key ID used is the unique transaction time stamp in combination with the identifier of the current user.

Creating an SQL Access Layer

The following topics are covered below:

- General Information
- Different Database Accesses with Adabas C and SQL
- Creating SQL Tables and DDMs
- Access to SQL Tables

General Information

Encapsulating the Database Accesses

Various database systems can serve as a basis for holding the data of an information object.

This is made easier by encapsulating the database accesses. Then it is only necessary to create a new access layer for each target database.

Access to an "object type" by the application remains as described in the preceding sections of this documentation. Modifications to the application programs are not necessary.

Creating an Access Layer

To create an access layer for an SQL database, for example, Adabas D, suggested codes are available. They are handled similarly to the suggested codes for the Adabas C access layer.

To support the largest possible number of SQL databases, the lowest common denominator must be found, that is, field lengths must be adjusted to the target platform with the most limitations. When SQL statements are used, they must not use database specific syntax features.

Definition of the Access Layer

The following sections describe the definition of the access layer from the viewpoint of redoing an existing Adabas C access layer.

They also contain notes on implementing a new system.

Different Database Accesses with Adabas C and SQL

Accesses to an SQL database differ in some ways from accesses to Adabas C data:

- Most differences are found primarily in the read loops which return multiple records.
- Single accesses can be reshaped easily.

These differences require a new access layer which can be largely derived from the Adabas C access layer.

The following topics are covered below:

- Converting a Sequential Read Access
- Converting a Single Access
- Creating Read Accesses
- Access using a Key with Several Components

- Inserting a New Record

Converting a Sequential Read Access

The conversion of a sequential read access involves use of the SQL ORDER BY statement.

Natural DML (Data Manipulation Language):

```
*
      RLI. READ <view name> BY <key> STARTING FROM <#start key>
*
        <further processing>
*
      END-READ
```

SQL Syntax:

```
*
      RLI. SELECT ALL * INTO VIEW <view name> FROM <table name>
           WHERE <key>  >=  <#comparison key>
           ORDER BY <key> ASC,
*
        <further processing>
*
      END-SELECT
*
```

Converting a Single Access

The counterpart of a single access to an object can be as follows:

Natural DML:

```
*
    FRE. FIND <view name> WITH <key> = <#start key>
*
         <further processing>
*
     END-FIND
*
     IF *NUMBER(FRE.) GT 0
         MOVE 1 TO PZ_AS_REC_EXIST    /* record exists
     ELSE
         MOVE 1 TO PZ_AS_RSP          /* see PZ_AS_REC_EXIST
     END-IF
*
```

SQL Syntax:


```
*
     FRE. SELECT SINGLE * INTO VIEW <view name> FROM <table name>
         WHERE <key> = <#comparison key>
*
     END-SELECT
*
     IF *COUNTER(FRE.) GT 0
*
         <further processing>
         MOVE 1 TO PZ_AS_REC_EXIST    /* record exists
     ELSE
         MOVE 1 TO PZ_AS_RSP          /* see PZ_AS_REC_EXIST
     END-IF
*
```

Creating Read Accesses

When creating the read access, you must ensure that, Adabas C-specific elements are converted.

Since periodic groups or multiple fields are not possible in SQL, these elements must be represented by corresponding division into tables and DDMs. The database design of a relational database has a flat structure, which means that the number of tables and relations increases in comparison to that of a design using Adabas C features.

See the following example for an illustration of what is involved.

- Language-dependent fields with several occurrences that are defined as keys are maintained in their own sub-table.
- Access to this sub-table is carried out through a language-dependent key which consists of a main key and an additional language identification.
- The language-independent information is read from the accompanying main table using the corresponding main key.

SQL Syntax:

```
*
     FREL. SELECT ALL * INTO VIEW <language view name> FROM <language table name>
         WHERE    <language key>    =    <#comparison language key>
         AND      <language>        =    <#comparison language>
*
         <further processing>
*
     END-SELECT
*
     IF *COUNTER(FRE.) GT 0
         MOVE 1 TO PZ_AS_REC_EXIST    /* record exists
     ELSE
         MOVE 1 TO PZ_AS_RSP          /* see PZ_AS_REC_EXIST
         ESCAPE ROUTINE
     END-IF
*
*
     FRE. SELECT ALL * INTO VIEW <view name> FROM <table name>
         WHERE <key> = <FREL.key>
                                    /* Reading the main object using the <key>
                                    /* of the language dependent value found
*
         <further processing>
*
     END-SELECT
*
```

Access using a Key with Several Components

If an SQL access uses a key that consists of several components, the counterpart to a READ statement in Natural DML can be implemented by an appropriate request.

```
RLIA. SELECT ALL * INTO VIEW <view name>
         FROM <table name>
         WHERE    <key-1>      >    <#comparison key-1>    OR
*
                  <key-1>      =    <#comparison key-1>
              AND <key-2>      >    <#comparison key-2>    OR
*
                  <key-1>      =    <#comparison key-1>
              AND <key-2>      =    <#comparison key-2>
              AND <key-3>      >=  <#comparison key-3>    OR
*
         ORDER BY <key-1> ASC ,
                  <key-2> ASC ,
                  <key-3> ASC
```


Inserting a New Record

The insertion of a new record is created using the available DDM and table fields. The individual fields of the DDM are transferred into the table by the appropriate SQL syntax .

```
*
*     Store record
*
      INSERT INTO <table name> (
         <field-1> ,
         <field-2> ,
         <field-3> )
      VALUES (
         <view name.field-1> ,
         <view name.field-2> ,
         <view name.field-3> )
```


Creating SQL Tables and DDMs

The communication of the access layer with the database uses DDMs even with SQL database systems.

The interface must be modified according to the target database.

When creating the tables and DDMs, the limitations of the target systems must be accommodated. These are:

- no multiple fields or periodic groups;
- no definition of super-/sub-descriptors;
- limitations in the field lengths differ from Adabas C.

The limitations lead to different DDMs. The following sections describe some of the possible modifications.

- Short Fields with Occurrences
- Long Fields with Occurrences
- Multiple Fields that are Descriptors
- Converting Formats
- Example of an Unsupported Field Format Conversion
- Defining Tables

Short Fields with Occurrences

Multiple fields that are not part of a descriptor and are shorter than 251 bytes are converted into individual fields with redefinitions.

DDM for Adabas C

```
V   1 <view name>
M   2 xxx_LNAME_LC    A    16   (1:9)
R   2 xxx_LNAME_LC
    3 xxx_LNAME_LC_G            (1:9)
    4 xxx_NAME_LC_LONG N    1
    4 xxx_NAME_LC A   15
```

DDM for SQL Databases with Longer Redefinition

```
V   1 <view name>
    2 xxx_LNAME_LC_R   A   144
R   2 xxx_LNAME_LC_R
    3 xxx_LNAME_LC     A   16   (1:9)
R   3 xxx_LNAME_LC
    4 xxx_LNAME_LC_G            (1:9)
    5 xxx_NAME_LC_LONG N   1
    5 xxx_NAME_LC A    15
```

This type of DDM is advantageous in that only a few fields must be defined.

Dialogs and programs access the variables via redefinition. Access to the SQL data is through the unredefined whole fields.

DDM for SQL Databases with Individual Fields Numbered


```
V   1 <view name>
    2 xxx_LNAME_LC_1   A   16
R   2 xxx_LNAME_LC_1
    3 xxx_LNAME_LC_G_1
    4 xxx_NAME_LC_LONG_1   N    1
    4 xxx_NAME_LC_1    A   15
```

This form of definition in a DDM has the disadvantage that many fields must be defined.

Access to all variables and SQL data is through the individual fields. The redefinitions here allow access to individual components.

Long Fields with Occurrences

Multiple fields or periodic groups that are not part of a descriptor but, including all occurrences, are longer than 251 bytes, can be converted into large individual fields with the redefinition of the group.

The allocation of the variables is through a redefinition.

The access to the SQL data is through the individual whole fields.

DDM for Adabas C

```
V   1 <view name>
P   2 xxx_field   A   80   (1:9)
```

DDM for SQL Databases

```
G   2 xxx_field_G
    3 xxx_field_1 A   240
    3 xxx_field_2 A   240
    3 xxx_field_3 A   24
R   2 xxx_field_G
    3 xxx_field   A   80   (1:9)
```

When processing multiple fields, conversion must be carried out in the same way.

To remove blank entries within the array, an appropriate routine must be written.

Multiple Fields that are Descriptors

Searching within a multiple field is possible with Adabas C. The SQL-specific conversion must then convert the multiple field by defining a main table and a sub table.

A second DDM consists of the fields via which a read access is possible.

Redundant holding of data is avoided since the information in the first DDM contains only the non-descriptor components. If a record is read through the multiple field, the data of the first view of the main table must be read through the common main key.

DDM for Adabas C


```
V   1 <view name>
    2 xxx_ID A    4
    2 xxx_FIELD1  A   20
    <further fields>
*
M   2 xxx_LNAME   A   16   (1:9) /* Descriptor
R   2 xxx_LNAME
    3 xxx_LNAME_G         (1:9)
    4 xxx_NAME_LONG   N    1
    4 xxx_NAME   A    15
```

DDM for SQL Databases

```
V   1 <view name>
    2 xxx_ID A    4
    2 xxx_FIELD1  A   20
    <additional fields>


V   1 <view name 1>
    2 xxx_ID A    4
    2 xxx_LONG    N    1
    2 xxx_NAME    A    15
```

Converting Formats

Some formats must be converted into a corresponding SQL target database format. Here again, a common denominator must be used.

The conversion of unsupported field formats is described in the section below.

Example of an Unsupported Field Format Conversion

In the following example, the conversion of a field not mappable in the length is performed.

In general, fields defined as numeric or integer are less problematic in a client/server environment than fields that are either packed or binary.

DDM for Adabas C

```
      1 xxx_ID P   20
```

DDM for SQL Databases

```
      1 xxx_ID_R   A   20
R     1 xxx_ID_R
      1 xxx_ID N   20
```


Definition in the SQL Table

```
        xxx_ID   CHAR(20)
```

In all instances in which mapping is possible, a conversion to a simple format is carried out.

DDM for Adabas C

```
        1 xxx_ID P   12
```

DDM for an SQL Database

```
        1 xxx_ID N   12
```

SQL Table

```
       xxx_ID    DEC(12)
```

Defining Tables

How is a Table Created with DDMs?

The DDMs must be rewritten in a corresponding SQL database table definition.

The new definitions created are entered as SQL tables in the target database.

The conversion of unsupported formats depends on the target database.

The definition of the individual SQL table matches the DDMs that were converted for an SQL database.

DDM for Adabas C

```
  V   1 <view name>
      2 xxx_ID  A   4
      2 xxx_FIELD1  A   20
      <further fields>
  *
  M   2 xxx_LNAME   A   16   (1:9) /* Descriptor
  R   2 xxx_LNAME
      3 xxx_LNAME_G          (1:9)
      4 xxx_NAME_LONG   N    1
      4 xxx_NAME    A   15
```

DDM for SQL Databases

```
V   1 <view name>
    2 xxx_ID  A   4
    2 xxx_FIELD1  A   20
    <further fields>


V   1 <viewl>
    2 xxx_LANGUAGE_ID  A   4
    2 xxx_LANG    N    1
    2 xxx_NAME    A    15
```

SQL Table

```
CREATE TABLE <table name>
(
    xxx_ID    CHAR(4),
    xxx_FIELD1    CHAR(20),
    <further fields>

)

CREATE INDEX <table name>.xxx_CODE

CREATE TABLE <table language>
(
    xxx_ID    CHAR(4),
    xxx_LANG  DEC(1),
    xxx_NAME  CHAR(15)
)

CREATE INDEX xxx_LNAME ON <table language> (xxx_LANG, xxx_NAME)
```


Access to SQL Tables

The following topics are covered below:

- Modifications of a Record
- Modifications of Individual Fields
- Optimizing Accesses
- Application of System Variables
- Allocation of Variables

Modifications of a Record

For the modification of an entire record, the field list in the form of the whole view is passed. The field list contains the complete record from the view.

The modification of the record is carried out with a Searched update.

Natural DML

```
FUP. FIND <view name> WITH <xxx_key> = <xxx_key_FROM>
     MOVE <#xxx_fields> TO <view name.xxx_fields>
     MOVE .....
     UPDATE
END-FIND
```

SQL Syntax

```
FUP. SELECT ALL COUNT INTO LZ_COUNT FROM <table name>
         WHERE <xxx_key> = <xxx_key_FROM>
END-SELECT
*
IF LZ_COUNT GT 0
    MOVE 1 TO PZ_AS_REC_EXIST    /* record exists
    MOVE <#xxx_fields> TO <view name.xxx_fields>
    MOVE .....
    UPDATE <view name> SET *
    WHERE <xxx_key>    = <xxx_key_FROM>
*
ELSE
    MOVE 1 TO PZ_AS_RSP          /* see PZ_AS_REC_EXIST
END-IF
```

Modifications of Individual Fields

The modifications of individual fields are also carried out with this read access. The field list, however, does not contain the entire record, but the fields that are actually modified.

SQL

```
UPDATE <table name>
     SET <xxx_field1> = <view name>.<xxx_field1>
WHERE <xxx_key>   = <xxx_key_FROM>
```

Example: Dating


```
UPDATE <table name>
    SET <xxx_EFD_INV>      = RUP.<xxx_EFD_INV>    /* modified value of the view
WHERE <xxx_ID>             = RUP.<xxx_ID>         /* key value found
    AND <xxx_EFD_INV>      = <#EFD_OLD>           /* date of record read
```

Optimizing Accesses

Using SQL syntaxes enables transparent access to the data.

The exact handling of the individual key components and the conversion of data types are easier to apply with embedded SQL statements.

The optimization is thereby more in sync on the target database system. When using SQL syntax, however, the common denominator - i.e. using ANSI/ISO-SQL - must be taken into consideration.

The optimization of the accesses is influenced by the:

- design of the tables;
- definition of the primary key and of indices;
- read accesses using SQL syntax.

For the individual accesses and table creation, database-specific optimizations must be taken into consideration.

In the following example, only one variant of an SQL-specific optimization is presented.

Table Definition

```
CREATE TABLE <table name>
(
    xxx_CLIENT_ID               CHAR(2),
    xxx_ID                      CHAR(12),
    xxx_NAME                    CHAR(30)

PRIMARY KEY (xxx_CLIENT_ID, xxx_ID
)
```

Access to the Table

```
SELECT ALL * INTO VIEW <view name>
    FROM <table name>
    WHERE     <xxx_CLIENT_ID>       >    <#xxx_CLIENT_ID>  OR
              <xxx_CLIENT_ID>       =    <#xxx_CLIENT_ID>
        AND   <xxx_ID>              >=   <#xxx_ID>
*
    ORDER BY  <xxx_CLIENT_ID>  ASC ,
              <xxx_ID>         ASC
```

In this case the access uses the primary key of the table which provides the best performance. With the corresponding number of indices that are created, further sorting sequences are mapped.

```
CREATE INDEX xxx_KEY ON <table name> (xxx_CLIENT_ID, xxx_ID, xxx_NAME)
```

Specifying the sorting sequence in the ORDER clause (in the above example) causes the SQL database system to use the primary key as the preferred access path.

This access can be optimized, when only a minimum of OR concatenations is used in the WHERE clause. As a result of a logical OR concatenation, the SQL database creates two internal lists, which must be validated against each other. This causes poor performance during multi-record accesses.

The optimized SELECT statement runs as follows:

```
SELECT ALL * INTO VIEW <view name>
    FROM <table name>

    WHERE     <xxx_CLIENT_ID>   =   <#xxx_CLIENT_ID>
      AND     <xxx_ID>          >=  <#xxx_ID>
*
    ORDER BY  <xxx_CLIENT_ID>   ASC ,
              <xxx_ID>          ASC
```

Application of System Variables

The application of system variables within the access layer is also modified. The available variables are limited by the use of SQL statements.

The variable *COUNTER can be used for checks. *NUMBER is no longer accessed. The variable **ISN* is not available.

Natural DML

```
FRE. FIND <view name> WITH <key> = <#start key>
    .....
END-FIND
*
IF *NUMBER(FRE.) GT 0
    .....
END-IF
```

SQL Syntax

```
FRE. SELECT ALL * INTO VIEW <view name>
    FROM <table name>
    WHERE <key> = <#comparison key>
    .....
END-SELECT
*
IF *COUNTER(FRE.) GT 0
    .....
END-IF
```

The number of records is established using another access. For this, a target variable is to be defined, into which the value is put.


```
DEFINE DATA LOCAL
.....
01 LZ_COUNT (I4)
END-DEFINE
*
FUP. SELECT ALL COUNT INTO LZ_COUNT FROM <table name>
         WHERE <xxx_key> = <xxx_key_FROM>
END-SELECT
*
IF LZ_COUNT GT 0
     .....
END-IF
```

Allocation of Variables

The allocation of the values from the DDM to the parameter variables of the access module can, in most cases, be implemented by a MOVE BY NAME statement.

If, with the transfer, a format or length is modified, the value must be set individually.

If, in the access layer, a conversion into several individual fields is required, these must be set.

It is also possible to define, in the DDM, a group over the individual fields. A redefinition in the corresponding variables with occurrences is thus possible in the local data area.

The transfer can then be carried out with a MOVE BY NAME. In the parameter data area, the definition then matches the view.

SQL View

```
V   1 <view name>
G   2 xxx_field_G
    3 xxx_field_1              A    240
    3 xxx_field_2              A    240
    3 xxx_field_3              A    240
R   2 xxx_field_G
    3 xxx_field                A    80  (1:9)
```

Parameter Data Area Definition

```
    1 <PDA group name>
    2 xxx_field                A    80  (1:9)
```


User Exits

This section describes the user exits available with the Natural application shell. With these user exits, standard functionality can be adapted to the requirements of your environment.

- Initializing Access Protection
- Initializing Application-Specific Data
- Default Start-up Processing

Initializing Access Protection

Description

This user exit sets up the desired access protection for the application. As a result of the access verification, the initialization of the command data is dependent upon the setting in the user exit.

In addition, this user exit can determine whether the object types Z_APPLICATION, Z_COMMAND, Z_FUNCTION and Z_OBJECTTYPE are individually verified through the system set up.

In this case, the user exit is called for each object type. The individual verification can be coded in the corresponding DECIDE statements. Response code PZ_RSP determines whether or not the object type is accepted in the initialization data.

Subprogram: ZXUSEC0N
Parameter: ZXUSEC0A

Parameters

Input/Output	Parameter Variable	Description
Input	PZ_USER_ID	User ID
Input	PZ_OBJ_TYPE	Type of object
Output	PZ_SECURITY	0 - Application without access protection 1 - Application with access protection
Output	PZ_SECURITY_DEFAULT	0 - Default is disallowed 1 - Default is allowed
Input	PZ_CLIENT_ID	Client ID of current application
Input	PZ_CMD_ID	Command ID to be verified
Input	PZ_CMD_TYPE	Type of command to be verified
Input	PZ_CMD_PARM	Parameter of command to be verified
Input	PZ_APPL_ID	Application ID to be verified
Input	PZ_OBJ_ID	Object ID to be verified
Input	PZ_FCT_ID	Function ID to be verified
Output	PZ_CHECK_CMD	0 - Commands not individually verified 1 - Commands individually verified
Output	PZ_CHECK_APPL	0 - Applications not individually verified 1 - Applications individually verified
Output	PZ_CHECK_OBJ	0 - Object types not individually verified 1 - Object types individually verified
Output	PZ_CHECK_FCT	0 - Functions not individually verified 1 - Functions individually verified
Output	PZ_RSP	0 - Objects allowed 1 - Objects disallowed

Initializing Application-Specific Data

Description

This user exit initializes the application-specific data in LDA ZXXGLOBL. It is here that the user environment LZ_GLOB_DATA_CUSTOM is initialized. Additionally, one can, for example, determine whether or not the icon-based navigation is started and with which application.

Subprogram: ZXUPROFN
Parameter: ZXUPROFA

Parameter

Input/Output	Parameter Variable	Description
Input/Output	PZ_PROF_DATA	Profile data

When this user exit is started, the data from PZ_PROF_DATA is transferred to the variables of the LDA ZXXGLOBL. Before this user exit terminates, the data is returned to PZ_PROF_DATA.

Default Start-up Processing

Description

This user exit executes an application-specific Start Processing. It is here, for example, that the first standard dialog is opened.

External Subroutine: ZXUINITS
Parameter: ZXUINITA

Parameter

Input/Output	Parameter Variable	Description
Input	PZ_RECEIVE	Standard PDA ZXXREC0A

When this user exit is run, parameter PZ_RECEIVE is copied in the group PZ_LOCAL. The variables for group PZ_RECEIVE must not be modified.

Business-Specification Descriptions

The following topics are covered below:

- General Usage
- Dialog Function
- Object View and Other Modules with Parameter Interface
- Other Reusable Modules
- Validations
- Information Objects
- Data Elements
- Modification History
- Descriptive Traits

General Usage

The description of an object contains a business specification. To avoid the redundant storage of information, as long as objects can be identified from their references, it is not necessary to describe how they are related to one another.

This section describes various types of descriptive objects as well as the related descriptive traits.

Dialog Function

- Definition
- Function summary
- Related information objects
- Pre-definitions
- Validations (unless documented separately)
- Selection help
- Lower-level dialogs
- Performance aspects (optional)
- Comments

Object View and Other Modules with Parameter Interface

- Definition
- Function summary
- Input/output parameters
- Variables used
- Validations (unless documented separately)
- Comments

Other Reusable Modules

- Definition
- Function summary
- Variables used
- Comments

Validations

- Definition
- Related information objects
- Validations
- Comments

Information Objects

- Description
- Set structure
- Comments

Data Elements

- Description
- Value range
- Validation
- Access protection
- Comments

Modification History

At the beginning of the description, each object must have a modification log indicating who updated the object, when it was updated, and what modifications were made.

User	Date	Comment
USERX	11.06.95	Function completed
USERY	21.12.95	Program call XYZ adapted to new structure

Descriptive Traits

Descriptive traits are explained detail below:

Access Protection

Any data element related access restrictions.

Comments

Additional comments.

Definition

A brief explanation regarding the objective and content of the module.

Description

Module documentation.

Information Objects

All information objects which are accessed (read/write) from this object.

If access to an information object is performed using an object view, each information object of the object view must be listed together with the read or write operation codes.

Otherwise, all Natural views must be listed via which read/write access is performed from this object. In addition, a descriptor is to be provided.

A descriptor which is constructed from many components must be provided with the components in physical order.

The listing of Natural view and descriptor can be added at the earliest following the database design.

A tabular representation is recommended for readability:

Information Object		
L/S	Object view/	Oper. Code/Access Key ID

Input/Output Parameters

All parameters which are used when a module is called as well as all parameters which are returned by the module.

Lower Level Dialogs

Description of subfunctions which can be called locally from the main dialog function.

An indication as to whether the lower level dialog must be implemented as a modal or non-modal module must be provided.

If the subdialog is only available in one dialog function, the description of the subdialog can be provided here.

A reusable subdialog must be defined and described as a separate object and only listed here.

Pre-definitions - Defaults, Initializations

Defaults for dialog fields can be divided into two groups:

- Default entries when adding a data record;
- Calculations/derivations when adding/modifying a data record.

The calculation/derivation of field contents can result from:

- Algorithms (the formula used must be provided here);
- Contents of other information objects (these information objects must be listed);
- Profile definitions.

Representation in tabular form is recommended.

Performance Aspect

Information relating to performance aspects of the dialog function (response time behavior), for example:

- How often will the dialog function be invoked?
- How many users are expected to be using the dialog function concurrently?
- Will large amounts of data be processed, or only a single record?

Performance Scope

The performance scope must be clearly and precisely stated, i.e., which business functions are provided by the module.

Functionality which already is covered via frames must not be documented here (e.g., paging, transaction protocol).

Validation

For the specification phase, it is recommended that all known dialog related validations be described here, as long as these are not described in separate modules.

Formal checks on individual dialog fields, for example, ’Mandatory Field’ or ’Field must be completely filled’, can be included in the description of the data elements.

Later, in the implementation phase, all validations must be described centrally in the validation module of the object view, thereby making subsequent maintenance easier.

In this case, a reference to the validation module will be sufficient.

Selection Help

All data elements must be listed for which a selection help is to be provided.

If already known, the selected dialog element for the selection help can also be mentioned.

Set Structure

Information concerning set structure which will be useful for subsequent construction of database and application programs.

Used Variables

All variables which are used and a reference to where the variables are defined.

Value Area

If known, a listing of the possible values of the data elements.

